

BASIC XLTM

A Language For
Your ATARI[®] Computer



Precision Software Tools

30 DAYS TO UNDERSTANDING BASIC XL

**by BILL WILKINSON
and
DIANE GOLDSTEIN**

**This book is Copyright (c) 1983 by
Optimized Systems Software, Inc.
1221-B Kentwood Avenue.
San Jose, CA 95129**

**All rights reserved. Reproduction or
translation of any part of this work beyond
that permitted by sections 107 and 108 of the
United States Copyright Act without the
permission of the copyright owner is
unlawful.**

Preface

The computer age is upon us. From young children to senior citizens, the computer and the need for "computer literacy" have caught the attention and curiosity of many people. And, for better or worse, computer literacy has come to mean "learning to program in BASIC". However, learning to program is no easy task. Most books are written in computerese. You have to understand computers in order to understand the books.

Not so with "30 Days to Understanding Basic XL." While avoiding computer "jargon" as much as possible, this book is intended to introduce you to the fundamental concepts of programming a computer using the BASIC language.

BASIC (Beginner's All-purpose Symbolic Instruction Code) is an introductory computer language. Because BASIC uses many common English words, it is often considered the easiest of computer languages for the beginning programmer to learn.

Unfortunately, there are as many dialects of BASIC as there are of English. While this book focuses on the particular dialect known as BASIC XL, it nevertheless attempts to teach concepts universal to most computer languages.

Once you have mastered BASIC, you can then delve into the other mysterious computer languages like C, Action, and Pascal. For now, BASIC is the language you need to understand in order to do elementary programs on your ATARI Home Computer.

Contrary to what you may have read previously, it is our belief that not everyone is intended to be a computer programmer. However, even if you do not become a programmer, you should become aware of computers and how they work. Hopefully, the knowledge you gain from reading this book will make you feel more relaxed around computers.

By working through this book you will become more familiar with your ATARI Home Computer. In addition, you will learn common BASIC statements. This knowledge will permit you to use commercially produced products. You do not have to be a programmer to use programs developed by other people.

Finally, the intent of this book is to introduce you to BASIC XL and beginning computer concepts. In the process we hope to make you computer literate.

TABLE OF CONTENTS

page

7	INTRODUCTION	
9	CHAPTER I:	GETTING TO KNOW YOUR COMPUTER: ALL THE CONCEPTS THAT ARE FIT TO PRINT
		Cursor Keyboard {RETURN} Key Error Syntax {DELETE} {BACK S.} Statement PRINT Character String
14	CHAPTER II:	ARITHMETIC: COMPUTERS NEVER MAKE MISTEAKS
		+ - * /
18	CHAPTER III:	REMEMBERING NUMBERS: VARY VERY IMPORTANT
		Variable Numeric Variable LET
23	CHAPTER IV:	DIRECT MODE VS. PROGRAMMING MODE: THE BIRTH OF A PROGRAM
		Statement Direct Mode Program Line Number RUN LIST
28	CHAPTER V	A NEW BEGINNING: WIPING THE SLATE CLEAN
		NEW {CLEAR} {SYSTEM RESET} ; ,

- 33 CHAPTER VI REPETITION:
GETTING LOOPED

GOTO
Loop
{BREAK}
- 37 CHAPTER VII RELATIONAL OPERATORS:
IF YOU CAN PASS THE TEST

<
>
=
<>
<=
>=
IF
THEN
- 43 CHAPTER VIII INPUT:
TALKING BACK TO THE COMPUTER

INPUT Statement
- 47 CHAPTER IX LOGICAL OPERATORS:
DOES THIS MAKE SENSE?

AND
OR
- 51 CHAPTER X RANDOM:
I WON WHAT?

Random Selection
RANDOM
Integer
- 55 CHAPTER XI THE PROGRAM RECORDER:
HITS ON TAPE

Program Recorder
CLOAD
CSAVE
LIST "C:"
ENTER "C:"
RUN

59 CHAPTER XII THE DISK DRIVE:
BEING FLOPPY ISN'T SLOPPY

Disk Drive
Diskette
File Name
File Name Extension
Boot
DOS
System Diskette
DIR
LOAD
SAVE
ENTER
LIST

66 CHAPTER XIII THE PRINTERS:
HARDCOPY ISN'T HARD

Software
Hardware
Printer
Hardcopy
LPRINT
Output

69 CHAPTER XIV GRAPHICS, PART I:
I GET THE PICTURE

Graphics
Pixels
Graphics Window
Text Window
COLOR
PLOT
DRAWTO

76 CHAPTER XV EDITING FEATURES:
THE SCREENING PROCESS

DELETE
BREAK
CTRL
←- .
→
INSERT
↑ .
↓
TAB
SPACE BAR
CTRL 1
CTRL 2
ESC
CAPS/LOWR

- 85 CHAPTER XVI IF REVISITED:
THEN WE CAN DO ANYTHING

IF...THEN
END
:
- 88 CHAPTER XVII SUBROUTINES:
CALLING FOR HELP

Subroutine
GOSUB
RETURN
- 95 CHAPTER XVIII BETTER LOOPS:
WHAT'S THE NEXT STEP

FOR
NEXT
STEP
Negative Numbers
- 100 CHAPTER XIX FOR LOOPS REVISITED:
ANOTHER STEP UP

FOR
NEXT
Nest
- 106 CHAPTER XX STRING VARIABLES, PART I:
REMEMBERING WORDS

String Variable
DIM
\$
=
<>
- 111 CHAPTER XXI STRING VARIABLES, PART II:
EVEN WORDIER

Destination String
Source String
Substring
Subscripts
LEN()

120	CHAPTER XXII	SOUND: YOU CALL THAT MUSIC?
		SOUND Sound Channel Pitch Sound Quality Volume
124	CHAPTER XXIII	GRAPHICS, PART II: I CAN WRITE BIGGER THAN YOU
		Graphics Window Text Window PRINT #6; POSITION
127	CHAPTER XXIV	GRAPHICS, PART III: MY PICTURE IS IMPROVING
		Graphic Mode Graphics Window Text Window Pixels COLOR PLOT DRAWTO
134	CHAPTER XXV	GRAPHICS, PART IV: ALL THE COLORS OF THE RAINBOW
		Color Registers COLOR SETCOLOR Luminance Hue
142	CHAPTER XXVI	GRAPHICS, PART V: A REVOLUTION IN RESOLUTION
		GRAPHICS 8 Luminance Hue
145	CHAPTER XXVII	PROGRAMMING AIDS: COSMETIC SURGERY
		REM NUM RENUM

150 CHAPTER XXVIII THE JOYSTICK:
MANUAL OR AUTOMATIC

HSTICK
VSTICK
STRIG

154 CHAPTER XXIX A REAL LIVE VIDEO GAME:
SNAILS' TRAILS

163 CHAPTER XXX CONGRATULATIONS:
30 END

165 ANSWERS:

INTRODUCTION

Before you begin working in "30 DAYS to UNDERSTANDING BASIC XL" there are several details concerning the organization and the format of this book that you should understand.

Chapters I through V are introductory chapters, and they describe concepts that must be understood before any real programming techniques can be learned. Beginning programming methods are covered in Chapters VI through XV. Also, please note: in Chapters XII, XIII and XIV we explain some equipment that can be used along with your home computer. If you already own these devices, the explanation should enhance your understanding of how they operate. If you do not own any additional equipment, you should read these chapters to obtain information on support products available to expand the usage of your home computer. In Chapter XV we assume you are using a color monitor or color television screen as your computer display. If your ATARI Home Computer is not attached to a color display, please work through the chapter anyway. The only difference will be in the lack of color.

In Chapters XVI through XXI additional beginning concepts are explained. The last section of the book, Chapters XXII through XXVIII, further explores the graphics and sound capabilities of the ATARI computer. The last chapters, Chapters XXIX and XXX, are a summary of concepts covered and a final example. Our example is a relatively easy video game which will indicate what you can produce with a little creativity and the BASIC commands covered in this book.

To make the learning process easier for you, we have organized each chapter in the same manner. First, each chapter begins with a glossary. Study this before you start reading the main body of the chapter. The glossary provides an introduction to the terms and concepts covered within the chapter.

Following explanatory paragraphs are sections labelled "Instruction". Please complete these sections. Usually you will be asked to type commands or statements into the computer. Instruction sections are intended to give you hands-on experience or, in computer talk, give you experience "interfacing" with your ATARI computer system.

At the end of each chapter are exercises. The answers are also provided at the end of the book. These

exercises are intended to provide you with additional practice and to integrate new concepts with those previously explained.

In order to visually emphasize certain words and ideas we have employed several different print styles or fonts. First, {braces} indicate a key on the computer's keyboard. For example {RETURN} refers to the key on the keyboard and also means "push the key labelled 'RETURN'".

Sometimes one key may have two or more names and functions associated with it. In most cases we use only one of the names to designate a particular key.

When necessary we use bold type to emphasize key words and italics to give instructions or make commentary.

In some chapters we have enclosed additional information in boxes. These boxes are intended to provide the reader with supplementary information. Although interesting and informative, these boxes are ancillary to the explanations provided within the main body of the text.

Finally, this book is to be used with your ATARI Home Computer. Insert the OSS BASIC XL cartridge. Turn on the computer and your monitor. If you do not understand these instructions, please first refer to your ATARI Operators Manual. Let's begin.

CHAPTER I

GETTING TO KNOW YOUR COMPUTER: ALL THE CONCEPTS THAT ARE FIT TO PRINT

Glossary:

Cursor -----	The cursor is a white (or light blue) square on the monitor which indicates where the next letter or number you type will appear.
Keyboard -----	The area on your Atari Home Computer which contains letters, numbers, and additional special keys.
{RETURN} -----	The return on your computer keyboard. By hitting this key, you tell the computer that you have completed entering an instruction or an answer.
Error -----	A mistake; the Atari Home Computer will indicate an error when you have made a typographical mistake, misplaced punctuation, misspelled a word, or given a command or statement in a form which the computer does not recognize.
Syntax -----	The pattern, structure or rules which describe the language the computer understands.
{DELETE} -----	This key will "erase" any letter you type by accident; it moves the cursor one space to the left each time you hit the key, deleting the letter, symbol, or number it replaces.
{BACK S} -----	This means back space and is the same as {DELETE} as described in the above definition.
Statement -----	Is the action part of a computer language just as a verb is the action part of English.
PRINT -----	A common command that tells the computer you want to see something on the screen.

Character -----	Any number, letter, or special symbol on the keyboard.
String -----	Any group of characters inside quotation marks.

After you have properly connected your Atari Home Computer, placed the BASIC XL cartridge into the computer, and turned on the computer you will notice that the monitor displays the word READY. Located directly under the R is a white square. This square is the cursor. It indicates where the next letter or number you type will appear. A cursor is like a pointer, and it will move as you type, indicating where the next character will appear on the screen. Your Atari computer keyboard is similar to an electric typewriter's keyboard. It has some extra keys and some additional special features, but in general the letters and numbers are in the same location.

First, locate the {RETURN} key. You will find it on the right side, second row from the top. Press the {RETURN} key. By hitting {RETURN} you are telling the computer that you are there. Also, you are telling the computer that you are entering or have entered an answer or information. When typing on the computer's keyboard, usually nothing will happen until you press the {RETURN} key!

Your computer monitor or TV screen should say READY. (If it does not, check the Atari Operators Manual to be sure your computer is connected properly.) If you continue to press the {RETURN} key, the cursor will move down the left hand margin. If you press the {SPACE BAR}, the cursor will move toward the right hand margin.

Instruction:

You Type:

PINT
Press the {RETURN} key

Your computer should say
ERROR - PINT[]

([] is actually an inverse video space.)
This is a sample of a Syntax Error.

Syntax refers to the well-defined set of rules that comprises or makes up a computer language. Actually any language--including English--has such a set of rules: try to understand this, proper without syntax the. Did you have trouble understanding that? The computer isn't as smart as you, so you have to be even more careful about what you tell it!

In the example shown above, you have told the computer something it does not understand. Whenever you make a typographical error, misplace punctuation, or misspell a word, the computer will display the line you typed together with the word "ERROR". It will show you the location where it thinks your error is by reversing the colors of the character and background at that spot.

Please note: For now, if you make a typographical error, simply retype the line. If you notice the error before you have pushed {RETURN}, use the {BACK S} key. {BACK S} means "back space" and is located above the {RETURN} key. Use this key to back up to your error and then retype the line from that character to the end of the line. The complete Atari editing system will be discussed in Chapter 15.

```
-----  
| Notice as you type information |  
| into the computer and the |  
| computer responds, there is a |  
| great deal of data on your |  
| monitor or television screen. If |  
| the accumulation of data is |  
| distracting, press the {SYSTEM |  
| RESET}. This will clear the |  
| screen and the word READY will |  
| reappear. |  
|-----|
```

Instruction:

You Type:

PRINT

Press the {RETURN} key.

A statement is the active part of any computer language. Statements translate into an action you want the computer to perform. In BASIC there are many different statements. PRINT is a common statement and indicates that you would like to see some information on your screen.

Your screen should look like this:

```
READY
PINT
ERROR PINT[]
PRINT
```

READY

You have just told the computer to PRINT nothing, and it did exactly that; it PRINTed a blank line. If you want to see something on the screen you must tell the computer to PRINT.

```
-----
| Originally,      computers  were |
| attached to printers; there were |
| no monitors or  TV screens.  |
| Today, a better word might be   |
| "SHOW" or "DISPLAY". However,  |
| we're still stuck with PRINT.  |
|-----
```

Instruction:

You Type:

```
PRINT "HELLO"
Press the {RETURN} key
```

Your screen should look like this:

```
READY
PRINT "HELLO"
HELLO
```

READY

Any words or numbers which you enclose in quotations marks will be treated as a unit by the computer. Any group of characters--that is, letters or numbers inside quotation marks--is called a "string". When you tell the computer to PRINT a string, it does exactly that. Of course, the computer doesn't PRINT the quotation marks. After all, if you told someone to say "HELLO", you wouldn't expect them to say "QUOTATION-MARK HELLO QUOTATION-MARK", would you?

Being able to PRINT such quoted strings (which are also sometimes called 'literal strings' or some other similar name) is invaluable. Most dialogues with the computer will involve several PRINTs of strings.

Before you begin our practice exercises, here are some additional facts that might help you relax as well as learn computer programming:

- . Even the very best programmers make errors. Errors are part of the learning process. Don't let them upset you.
- . As you type, the computer will automatically continue or "wrap around" to the next display line. Even if something you type in occupies more than one display line, the computer treats the line or lines as a unit. However, you may not exceed more than three display lines without starting a new PRINT statement.
- . The only way the computer knows that you have finished a thought or entry is by pressing the {RETURN} key.
- . If you press the {RETURN} key before you have finished a thought--say at the end of the display line--the computer will execute what it can. It may, however, consider what you have typed to be illegal and give you an ERROR message.
- . It is normal for your computer to use UPPER-CASE LETTERS. Lower case is discussed in our chapter on editing.
- . If you want to clear the screen of all confusing characters, you may either turn your computer off and then back on, or you may press the {SYSTEM RESET} key.

Exercise:

1. Tell the computer to PRINT your name.
2. Tell the computer to PRINT a famous quotation like "GIVE ME LIBERTY OR GIVE ME DEATH".

Reminder: Don't forget to use quotation marks.

CHAPTER II

ARITHMETIC: COMPUTERS NEVER MAKE MISTEAKS

Glossary:

+	The symbol for addition which the computer understands.
-	The symbol for subtraction.
*	The symbol for multiplication.
/	The symbol for division.

Using BASIC, a computer is capable of performing simple arithmetic calculations such as: addition, subtraction, multiplication, and division.

Instruction:

You Type:

PRINT 3+7

Press the {RETURN} key

Your screen should look like this:

READY

PRINT 3+7

10

READY

Instruction:

You type:

PRINT 9-5

Press the {RETURN} key

What does your screen look like?

Notice: it is neither necessary nor correct to hit the equal sign key. The PRINT statement alone is sufficient to tell the computer to automatically perform the calculation and display the results.

The symbol for multiplication is the asterisk (*). If you want to try multiplying, then you must use the asterisk.

Instruction:

You Type:

PRINT 6*4

Press the {RETURN} key

The computer should reply with an answer of 24. But also try typing these lines:

PRINT 6X4

PRINT (6) (4)

Even though such forms are used in Algebra and other mathematics, they are not legal in BASIC. You must use an asterisk to cause BASIC to multiply.

The symbol for division is the slash.

Type: PRINT 24/2

Press the {RETURN} key

The computer will respond with an answer of 12.

Instruction:

You type:

Computer Responds

PRINT 72/9 {RETURN}
PRINT 7*8 {RETURN}
PRINT 2+4+6+8 {RETURN}
PRINT 3*21 {RETURN}
PRINT 3+4*2 {RETURN}

8
56
20
63
11

Were you surprised by the last response? Did you expect the answer to be 14? The computer normally performs arithmetic functions in left to right order, but all multiplication (*) and/or division (/) is performed before any addition (+) and/or subtraction (-). Some people call this the "MDAS rule" because the computer executes all multiplication and /or division before executing addition and/or subtraction.

Instruction:

You type:

Computer Responds

PRINT 850 {RETURN}	850
PRINT "850" {RETURN}	850
PRINT 2+6 {RETURN}	8
PRINT "2+6" {RETURN}	2+6

Does the last result surprise you?

Remember a string is any group of characters inside quotation marks. When confronted with a string, the computer treats the string as a unit. Literally, what goes in, comes out.

Instruction:

You type:

Computer Responds

PRINT "5+9*7 CATS=1,000" {RETURN}	5+9*7 CATS=1,000
PRINT "XQR/%#\$" {RETURN}	XQR/% # \$
PRINT "ADV*21970" {RETURN}	ADV*21970
PRINT "800-421-0009"	800-421-0009
PRINT 800-421-0009	370
PRINT "999-38-0101"	999-38-0101
PRINT 999-38-0101	860

In our examples above, the computer performed all arithmetic operations it confronted EXCEPT on those which were enclosed in quotation marks and just happened to look like operations. Since anything in quotation marks is a string, even arithmetic expressions enclosed with quotation marks become strings.

Notice it does not matter what you place in a string. When told to reconstruct the string, the computer does so exactly in the manner in which it was entered.

From now on, we won't usually show you when to press {RETURN}. Just remember to press the key when you

finish an instruction to the computer or when you give it an answer. If the computer appears to be doing nothing, press [RETURN]. In most cases, this will cause the computer to do something and then give you control.

Exercises:

1. Tell the computer to PRINT your name, address, and phone number.
2. Write a PRINT statement that will allow the computer to compute each of the following:
 - a) $15 + 25$
 - b) $38 - 14$
 - c) $680 * 12$
 - d) $25 / 5$
3. Write a PRINT statement that will display your social security number.

CHAPTER III

REMEMBERING NUMBERS: VARY VERY IMPORTANT

Glossary:

Variable Anything which has the capacity to change.
----- Examples include your age, the price of a
 stock, and the rate of speed at which you
 drive your car.

Numeric A variable which contains only numbers.
Variable

LET A statement which assigns a value to a
--- variable.

A variable is anything which is subject to change. The weather is a variable because from day to day and season to season, it is capable of changing. The prices of food and gasoline are also variables. Given certain conditions, the price of any given item might change. These are variables.

To better understand the concept of variables as used in OSS BASIC XL, imagine a teacher standing at the door to her classroom the first day of the new year in an elementary school.

The first student arrives; she hands him a card with his name on it and assigns him the first cubby hole to store his possessions. He tapes his name to the cubby, and then he puts his jacket inside the cubby.

The next student arrives and goes through the same procedure taking the second cubby, but she has a jacket, notebook, umbrella, and a lunchbox to put into her cubby.

As each student arrives, the next succeeding cubby is assigned; each cubby is filled or not filled depending on what the student has brought to school that day. Anything which is placed in the cubby becomes associated with that particular cubby hole, until something new is added to the cubby.

In computer talk, a "value" is stored in a "variable". Any child could have walked into class first and thus been assigned any "variable" cubby. Also, each child

could have brought any number of items which became the "value" in the variable.

Each variable cubby is assigned a name. The name was chosen in a meaningful manner, using the child's name for identification. Computer variables function much like our "cubby" example.

Although there are different kinds of variables, each numeric variable is represented by a name. In BASIC XL, variable names always begin with a letter. A numeric variable name may consist of one letter or a combination of letters or numbers. Examples of legal variable names are: N, X, STOCKPRICE, AC, or B3. There is no practical limit (other than the line size) on the length of a name. Some illegal variable names are: lABC, A\$, or A]B.

Although a single letter is sufficient to identify a variable, it is helpful to use variable names which are more descriptive. For example, the single letter I is sufficient and acceptable as the name of a variable associated with income. However, it is just as easy and more meaningful to use INCOME as the name of the variable for the value associated with income. As you do more programming and as you use many variables within the same program, the value of using meaningful names for variables will become more apparent.

It is important to remember that each numeric variable may hold a single number at any given time. Although the value of a variable may change several times during the course of a program, the variable can represent only one value at a time.

In order to tell the computer you are going to use a variable you must use a LET statement. A LET statement assigns a value to a variable. The form of a LET statement is:

LET name of variable = value of variable

Instruction:

You Type:

Computer Responds:

LET X = 14

PRINT X

14

LET X1 = 35

PRINT X1

35

```
LET APPLES = 90
PRINT APPLES
```

90

```
LET R = 9
PRINT R
```

9

In the above examples values were assigned to different variables. The computer stores the value of the variable in its memory. When given additional instructions, (e.g., PRINT) the computer recalled the value of the variable and used it (e.g., PRINTed its value).

Instruction:

You type:

Computer Responds:

```
LET A = 5
PRINT A
```

5

```
LET A = 7
PRINT A
```

7

Notice in the examples above that the value of the variable A changes from 5 to 7. Whenever the value of a variable changes the computer remembers only the last value. We could continue to change the value of our variable A, but the computer would remember only the most recent value and none of the previous ones.

You will recall from Chapter 2 that it was possible to use more than one number in a PRINT statement as long as the numbers were connected by an arithmetic operator. The same is true for variables. A variable and a number may be combined in a PRINT statement as long as they are connected by an arithmetic operator.

Instruction:

You type:

PRINT A+1

8

(Remember, A had a value of 7, so the above statement is equivalent to typing PRINT 7+1.) In addition, variables may be used in conjunction with other variables and numbers as long as they are used in an arithmetic expression using arithmetic operators.

```
LET B = A+1
PRINT B
```

8

Although we have named a new numeric variable B, the computer still remembers the value of the variable A. In the above example, we have asked the computer to add 1 to the value of A and to let that new value (7+1) be the assigned as the value of the variable B.

Although $A+1 = B$ is an acceptable expression in algebra, it is illegal in BASIC XL. Remember, computers are not as smart as people. To a computer $B=A+1$ and $A+1=B$ are two very different expressions. The former is permitted in a program; the latter will produce an error. The statements which assign a value to a variable, MUST follow this form:

LET variable=expression.

Instruction:

You Type:

Computer Responds:

```
LET A=B
PRINT A
```

8

Now, we are telling the computer to let the value of the variable A be the same as the value of the variable B. Notice that the computer remembers only the new value of the variable A; the old value of A is completely disregarded.

We could also change the value of the variable in another way.

Instruction:

You Type:

```
LET F = 15
LET X = F
PRINT X
```

15

The computer stores the value of a variable in its memory. Each time you change the value, the computer also changes the value of the variable.

The really neat thing about a variable is its ability to change. During the course of a program, the value of a variable can change as many times as you would like.

The use of variables will become more apparent when we discuss loops. But for now, remember that a variable is anything that can change, and that the purpose of using a variable is to put the burden of remembering on the computer and not on your brain. For example, if you wanted to keep track of the days in a year, how many are left, and how many have already passed, you could keep track by using math and lots of paper. However, an easier and quicker way would be to use the computer.

Exercise:

- 1) What do the following LET statements do?

LET statements:

- a) LET X = 15
- b) LET B = 9 * 8 * 2
- c) LET F2 = 111 * 3 + 5
- d) LET Z = F2 - 15

- 2) Assign the sum of seven and five to a variable called TOTAL. Display the value of a number one greater than that sum.

- 3) Try the following:

LET GROSSPAY = 40 * 8

LET DEDUCTIONS = 122

LET NETPAY = GROSSPAY - DEDUCTIONS

PRINT NETPAY

What does the computer display?

CHAPTER IV

DIRECT MODE VS. PROGRAMMING MODE: THE BIRTH OF A PROGRAM

Glossary:

Statement -----	Is an instruction to the computer. An example is PRINT as used in chapters 1, 2, and 3.
Direct Mode -----	Allows the computer to respond immediately to any instruction you give it.
Program Line -----	Is any valid combination of symbols, values and statements entered into the computer. It instructs the computer to do a task.
Program -----	Is a series of one or more statements which tells the computer exactly what task it is to perform.
Line Number -----	Is a one to five digit number entered at the beginning of a program line. Note that an absence of a line number implies an instruction which is executed in direct mode.
RUN ---	Is the command which tells the computer to execute the program.
LIST ----	Is a command which tells the computer you want to see the program lines it has stored in its memory.

Up to this point, we have been using the computer in "direct" mode. That is to say, when we gave the computer a command, we wanted the response immediately. Examples of statements includes:

```
PRINT "HELLO"  
PRINT 7*964.
```

A computer program is a series of instructions which tells the computer to perform a sequence of tasks that hopefully will produce a desired output. These instructions are called statements. LET X = 15 is a LET statement. As stated earlier, a LET statement allows you to assign a number value to a variable. These are instructions for the computer.

It is important to tell the computer that you want to give it a series of commands before it executes your orders. Think about sending someone to the supermarket. You need ten items; but as soon as you mention the first one, the person runs out the door. He comes back five minutes later with your one item. You explain to him you need several products. This time he leaves after you have mentioned three items. When he returns, you are still missing six items. It would have been more efficient if he had waited for the entire list in the beginning.

The efficient way in which to tell the computer you have a series of commands for it to execute is by assigning a line number to each statement (command). When you type 10 or 20 or 30 before a statement, it tells the computer that more commands might follow.

By using line numbers, you take the computer out of direct mode and put it into programming mode. You do not have to turn any knobs or change anything with the computer; you just have to type a line number before each statement. Each line of a BASIC program must have a number. The computer executes statement lines in numerical order, regardless of the order in which they were typed into the machine. Even if you type in line 30 first and then type in line 10, the computer will act on line 10 first. In addition, you should know that it is often customary--but certainly not necessary--to number lines in increments of 10.

One reason for numbering program lines in increments of 10 is to allow for insertions. If you begin your program with the line number 10 and continue to number each line in increments of 10, you will have enough room to add a line anywhere in the program. Perhaps you forgot a line in the beginning; your new line could be assigned line number 5 or 6. Remember, the order in which you enter a line is not important. What is important to the computer is the number of the line.

Once you begin to use line numbers, the computer will not execute the lines with numbers until you type RUN. The command RUN indicates to the computer to begin processing the statements you have entered. Remember, it normally processes the lines in numerical order.

Instruction:

You type:

Computer Responds:

PRINT "PROGRAMMING IS FUN"

PROGRAMMING IS FUN

Now type:

10 PRINT "PROGRAMMING IS FUN"

Nothing happens until you also type RUN!

So now type RUN and press the {RETURN} key.

PROGRAMMING IS FUN

Instruction:

You type:

Computer Responds:

10 LET A = 1
20 PRINT A
30 LET B = A + 1
40 PRINT B
50 LET C = B + 1
60 PRINT C
RUN

1
2
3
READY

As mentioned earlier the computer will act on the lines in line number order, not necessarily the order in which you entered the lines. Once you start using line numbers, you are out of direct mode and in programming mode.

For example, sometimes as you are programming you may forget a line.

Instruction:

You Type:

Computer Responds:

10 LET R = 1
20 PRINT R
30 LET S = 10
40 LET T = 15
50 PRINT T
60 LET V = R + T
70 PRINT V
RUN

1
15
16
READY

Note that nothing happened with S! So let's insert another line.

You Type:

35 PRINT S

Now your program is out of order on the screen. However, if you were to RUN the program, the computer would PRINT:

1
10
15
16

READY

In order to get the computer to show you the program in the correct order, type LIST and hit the {RETURN} key.

You Type:

LIST {RETURN}

Computer Responds

10 LET R=1
20 PRINT R
30 LET S=10
35 PRINT S
40 LET T=15
50 PRINT T
60 LET V=R+T
70 PRINT V

Now your program is in the correct numerical sequence on the screen.

Remember, the order of entry of the lines does not matter to the computer. Once again, the computer will always act on the program in line number order, not the order in which the lines were typed into the machine.

Before you begin the exercises below, please turn off your computer. This will erase the computers memory. The next chapter will explain more about clearing the screen and the computers' memory.

Exercises:

- 1) Write a program which assigns a value to two variables. Then have it PRINT out the sum of the two variables and their product. Be sure to LIST and RUN your program.
- 2) Turn your computer off and then on again and type in the same program (as in exercise 1) again. This time, though, leave out the line which assigns a value to the second variable. LIST and RUN the program. What value does the computer assume for the unassigned variable?
- 3) Without turning off the computer reinsert the line which assigns a value to the second variable. Remember this line should now be the second line of your program. What happens when you LIST and RUN?

Chapter V

A NEW BEGINNING: WIPING THE SLATE CLEAN

Glossary:

NEW ---	A command which tells the computer to erase the portion of its memory that stores BASIC programs.
{CLEAR} -----	A key that when pressed at the same time as the shift key will erase the screen BUT NOT THE MEMORY.
{SYSTEM RESET} -----	A key that when pressed will clear the screen and place the computer back in direct mode.
; --	Punctuation for the PRINT statement which tells the computer to PRINT more on the same line instead of going to the beginning of the next line.
, --	Allows the programmer to PRINT words or numbers in columns.

All the information you have typed into your Atari Home Computer in program mode is stored in its memory. In a program each line number must be distinctive. No two program lines can share the same number. If you use the same program line number, the computer remembers only the one which was most recently entered. To see how this works, try this program:

```
10 PRINT "THIS IS THE ORIGINAL LINE 10"  
10 PRINT "THIS IS THE NEW LINE 10"  
RUN
```

The computer should respond:

```
THIS IS THE NEW LINE 10.
```

```
READY
```

If you have been working with this book and have not turned off your computer between Chapters IV and V, you may have found additional information PRINTed on your screen. If so, don't worry about it. That is what this chapter is all about.

Once you type in a program, the computer will normally remember it until you turn the power off. A more convenient way in which you tell the computer to "forget" everything it has stored in its memory is by typing the command NEW.

When you begin a new program, it is a good idea to first type NEW. This command erases the computer's memory. By typing NEW you avoid having your new program mixed in with your old program. Try this example:

Instruction:

You type:

```
NEW
10 PRINT "Y"
20 PRINT "E"
30 PRINT "L"
40 PRINT "L"
50 PRINT "O"
60 PRINT "W"
RUN
```

Does the computer do what you expected?

Now type:

```
10 PRINT "B"
20 PRINT "L"
30 PRINT "U"
40 PRINT "E"
RUN
```

Notice that O W is PRINTed after B L U E. Why did that happen? The extra letters were PRINTed because lines 50 and 60 from the last program are still in the computer's memory. There are two things you can do to avoid this problem. First, you could simply type "NEW" and press {RETURN} into the machine and then retype lines 10 through 40. Remember, though, once you type "NEW" you can NOT get a program back. Turning off your computer or opening the cartridge door will also erase your program from the computer's memory.

However, you CAN save the program and just eliminate the excess lines. To do this, just type the line number you want to delete and press {RETURN}. In the example above to get rid of the O W following B L U E, type,

```
50 {RETURN}
60 {RETURN}
```

Now type LIST and RUN. Notice that lines 50 and 60 are no longer in the computer's memory and the O and W are not PRINTed.

Also, please note, there is a difference between clearing the screen and clearing the computer's memory. To clear the memory type "NEW." To clear the screen, press the {CLEAR} key while holding down the {SHIFT} key. Or, you can also clear the screen by pushing the {SYSTEM RESET} key.

CAUTION! If a program is RUNning when {SYSTEM RESET} is pushed, the program will halt but will remain intact.

In addition to clearing the screen, there are other things that you can do to change the manner in which items appear on the screen. A PRINT statement automatically returns the computer to the left-hand margin. You can suppress the return by putting a comma or a semicolon after the last value in the PRINT statement.

For example:
10 PRINT 8*3;
20 PRINT 956,
30 PRINT "CARS"

The only difference between the comma and the semicolon is the semicolon tells the computer to stay on the same line while the comma tells the computer to stay on the same line and move to the next columnar position. (Columnar positions are generally 10 characters apart, but this can be changed by the user and is described in the OSS BASIC XL Reference Manual).

Remember, if you want information PRINTed adjacent and on the same line, use a semicolon(;). However, if you want information PRINTed in columns on the same line, use a comma.

Instruction:

You type:

Computer Responds

NEW
10 PRINT 10*7,
20 PRINT 5+2;
30 PRINT "SEE"
RUN

70

7SEE

It is also possible to have the computer display more than one item using a single PRINT statement. The items to be displayed may be separated by either a comma or a semicolon. As you might expect, the comma will cause the items to be PRINTed in columnar form. The semicolon will cause the items to be PRINTed adjacent to each other.

Instruction:

You type:

Computer Responds

NEW

```
10 LET A=12
20 LET B = A + 3
30 LET A = B
40 PRINT "THE VALUE OF A IS " ; A
RUN
```

THE VALUE OF A IS 15

Type:

Computer Responds

Now change line 40 to:

```
40 PRINT "The Value of A is" , A
RUN
```

THE VALUE OF A IS 15

Notice the difference a single punctuation mark makes. Although not as smart as people, computers are very consistent. In BASIC XL, the computer will always respond to statements using semicolons and commas in the same way.

Exercises:

1. Write a program which will PRINT the values of two different variables. Have the computer PRINT the values adjacent to each other on the screen.

Suggestion: Once you enter the answer to exercise 1, you need only to change one line. The lines in which you set the value of your variables may be reused for all three exercises.

2. Modify the above program so that the two values will be PRINTed about 10 spaces apart.

3. Modify the above program so that the sum (+), the difference (-) and the product (*) of the values of the two variables are printed on the same line.

Chapter VI

REPETITION: GETTING LOOPED

Glossary:

GOTO -----	A statement which tells the computer to jump or branch to a specified line number and then continues to follow the program instructions in the usual line number order.
Loop -----	The repetition of one or more statements.
{BREAK} -----	A key which when pressed stops whatever the computer was doing.
Initializing a variable -----	Giving the first or the initial value to a variable.
Counter -----	Keeps track of or counts the number of times a loop has been executed.
X = X + 1 -----	A non algebraic expression that allows a variable to act as a counter.

The commands or statements we have discussed so far have permitted the computer to work in a "straight line" sequence. First do line 10, then line 20, etc. In order to get the computer to repeat a process, you need to use a GOTO statement. This statement must be followed by a line number, and this instruction tells the computer to jump or branch to the line number following the word GOTO. For example:

```
10 PRINT "My Name is Sally"
20 GOTO 10
```

The computer will PRINT continuously MY NAME IS SALLY. The computer executes line 10 then reads line 20. It goes back to 10 then back to line 20 etc. The result is a "loop".

A loop is the repetition of a sequence of instructions. An "endless loop" is one in which the computer continuously loops back through the instructions and never ends. To stop an endless loop press the {BREAK} or {SYSTEM RESET} key.

A GOTO statement is VERY useful; it allows the computer to execute the same command, or set of commands, under various circumstances.

```
-----  
| A computer language purist might |  
| tell you not to use the GOTO    |  
| statement. It is true that there |  
| are other commands in BASIC XL   |  
| which produce the same results as |  
| GOTO (but in what is considered a |  
| more structured and "elegant"    |  
| manner). However, these other    |  
| commands are more advanced and    |  
| require more background knowledge |  
| in programming than we have      |  
| presented up to this time.       |  
-----
```

Instruction:

You type:

Computer Responds

```
NEW  
10 PRINT "N", "N-Squared"  
20 LET N = 1  
30 PRINT N, N*N  
RUN
```

```
      N      N-SQUARED  
      1      1  
READY
```

In analyzing this program, line 10 should not have been a problem. We simply asked the computer to PRINT or write on the screen "N" and, about 10 spaces later, "N-SQUARED".

In Line 20, we have set a value for our variable N. This is called initializing or giving a starting value to a variable. Notice that this statement is executed once.

Again, in Line 30, we have asked the computer to PRINT or show the value of "N" and the value of "N*N" on the screen. Since we have assigned a value to "N", the computer can execute these 3 lines as a program.

Instruction:

You type:

```
40 LET X = N + 1
50 LET N = X
60 GOTO 30
RUN
```

When you've seen enough, press the {BREAK} key.

In Line 40, we have identified another variable called "X". Furthermore, we have said "LET X = N + 1". You should read that as: Let the value of X equal whatever the value of N is plus 1 more. In other words, whatever the value of N is, the value of X is equal to one more than N.

Line 50 indicates that N is to be given a value equal to the current value of X. This is to say: first we decided that N would have a value of one. Then, X was assigned a value of N + 1 or 2. Now we are saying that really we want the values of N and X to be the same.

All of this is done so that when the computer gets to Line 60 and then in turn goes back to Line 30, the value of N changes. Each time the computer executes Line 30, the value of N changes. This is a counter. Essentially, the computer counts each time it has executed Line 30.

An easier way to write our program is:

```
READY
10 PRINT "N", "N-SQUARED"
20 LET N = 1
30 PRINT N, N*N
40 LET N = N + 1
50 GOTO 30
```

Look at our new line 40. How can $N = N + 1$? This is NOT an algebraic equation! What is meant here is LET the New value of N become the Old value of N plus one more.

```
      LET N = N + 1
(New value)  (Old Value)
```

Exercises:

- 1) Write a program which will display the number, the square of the number, and double the number.
- 2) Write a program that will make the computer count from one to forever.
- 3) Modify the above program so that it will PRINT your name and count at the same time.
- 4) Modify the above program so that it will PRINT your name and count at odd numbered increments only.

Hint: LET X = X + 2

- 5) Modify the above program so that it will begin counting with 5 and increases in odd increments.

Remember with these programs, when you have seen enough, press the {BREAK} key. Also since exercises 3, 4, 5, are modifications of exercise 2, just retype the particular line that needs to be changed. Be sure to LIST the program and then RUN it.

Chapter VII

RELATIONAL OPERATORS: IF YOU CAN PASS THE TEST

Glossary:

Relational Operators

< less than

> greater than

= equal to

<> not equal to

<= less than or equal to

>= greater than or equal to

IF a command which tells the computer to test
-- the truth of a comparison.

THEN a special key word which always follows an
----- IF comparison. When IF finds a true
 condition, the statement(s) following the
 word THEN are performed.

A computer does only what you specifically tell it to do. It cannot make decisions "on its own". However, as a programmer, you can formulate statements in such a way that the computer will be forced to make certain comparisons. Relational operators allow you to test two values and to determine what relationship one has to the other. The computer will then decide if the relationship or comparison is true or false. The relational operators most commonly used in BASIC are:

<	(less than)
4 < 6	read as: Is four less than 6?
200 < 400	read as: Is two hundred less than four hundred?
X < C	read as: Is the value of X less than the value of C?
>	(greater than)
9 > 5	read as: Is nine greater than 5?
15 > 0	read as: Is fifteen greater than zero?
Y > X	read as: Is the value of Y greater than the value of X?
=	(equal to)
4 = 2 + 2	read as: Is four equal to two plus two?
9 = 4 + 3 + 2	read as: Is nine equal to four plus three, plus two?
A = B	read as: Is the value of A equal to the value of B?
<>	(not equal to)
5 <> 10	read as: Is five not equal to ten?
A <> B	read as: Is the value of A not equal to the value of B?
DOG <> CAT	read as: Is DOG not equal to CAT?

For those of you who have studied algebra, the above symbol is probably a new one. <> is the same as \neq ; however, the computer will understand only <> since \neq is not available on the computer keyboard.

<=	(less than OR equal to)
19 <= 20	read as: Is the value of nineteen less than or equal to the value twenty?
R <= D	read as: Is the value of R less than or equal to the value of D?
N <= 1	read as: Is the value of N less than or equal to the value one?

>=	(greater than or equal to)
1 >= 0	read as: Is one greater than or equal to 0?
H >= 4	read as: Is the value of H greater than or equal to four?
E >= L	read as: Is the value of E greater than or equal to the value of L?

Again, the last two symbols are the closest the computer can come to the conventional math symbols.

In programming in BASIC, the IF statement allows you to set up comparisons using the relational operators. The format of an IF statement requires the key word THEN to follow a relational comparison. If the computer determines that the comparison is true, it will perform whatever instructions follow the THEN.

For example:

```
10 IF X > 5 THEN PRINT X
```

In the above program line, the computer will PRINT the value of X if it determines that X is greater than 5. If the computer concludes that the value of X is not greater than 5, the computer will drop down to the next program line. If none exits, it will do nothing else.

```
10 IF Y < 4 THEN GOTO 60
```

If the computer, in the example above, finds the value of Y to be less than 4, it will proceed to line 60. If the value of Y is not less than 4, the computer will drop down to the next consecutive program line.

Please note: Do not divide the two-character relational operators (<=, >=, and <>) with a space. If you do, the computer will list out an error.

Instruction:

You type:

```
10 LET F = 1
20 LET H = 6
30 IF F + H < 5 THEN PRINT "SUM IS LESS THAN 5"
40 IF F + H > 5 THEN PRINT "SUM IS GREATER THAN 5"
RUN
```

How does the computer respond? What does it PRINT? In the above program "SUM IS LESS THAN 5" will never be

PRINTED. Given the values of the variables F and H, the computer will always determine that F plus H is greater than 5. Now, try other values for F and H for example, (Try F = 1 and H = 1) by typing in a new line 10 and/or line 20.

You will recall that in Chapter VI, we discussed the GOTO statement and the loop. We can now use these concepts along with our relational operators to produce for the first time an example with all the classical elements of a true computer program.

Instruction:

You type:

```
NEW
10 LET G = 1
20 PRINT "HI, MOM"
30 LET G = G+1
40 IF G < 8 THEN GOTO 20
50 PRINT "END OF PROGRAM"
RUN
```

Notice that the computer has PRINTED "HI,MOM" seven times.

What is the process the computer follows? First, the computer reads line 10 and initializes the value of the variable G at 1. Then, the computer executes line 20. It PRINTS "HI, MOM". When the computer reaches line 30, it changes the value of the variable G by adding 1 to the old value of G as we did in Chapter VI. The computer then replaces the old value of G with its new value, 2. Each time the computer reaches line 40 it "tests" or "decides" if the value of G is less than 8. If the value of G is less than 8 the computer determines that the rest of the statement must be performed. It therefore obeys the command and loops back to line 20. On the other hand, if the value of G is not less than 8, the computer will determine that the rest of the statement should not be executed. Anytime an IF statement determines that the following comparison is false, it causes control to pass to the next succeeding line number. Therefore, in this case, control passes to line 50.

Also, it is important to understand that if you wanted "HI,MOM" PRINTed exactly 8 times, you would have to change line 40 to read:

```
40 IF G <= 8 THEN GOTO 20
```

or

```
40 IF G <> 8 THEN GOTO 20
```

or

```
40 IF G < 9 THEN GOTO 20
```

Change line 40 to be the same as one of the lines 40 above. (Remember: just retype the line including the line number. The computer will automatically disregard the previous line 40.) RUN the resultant program and verify that the loop now executes exactly 8 times.

Try changing line 20 to:

```
20 PRINT "HI, MOM", G  
RUN
```

Remember, by using the comma, we force the computer to display HI, MOM and the value of G on the same line one column apart.

Now change line 30 to:

```
30 LET G = G + 2  
RUN
```

How does this affect the program? How many times does the computer display HI, MOM and the value of G? What value does it display?

Change line 10 to say:

```
10 LET G = 100
```

What happens to the program? Why is HI, MOM and the value of G displayed only once? Because we initialized the value of the variable G at 100, the value is already greater than the test value in line 40.

Exercises:

- 1) Write a program which will count to 100 by fives and display the results.
- 2) Write a program which will PRINT this message 10 times "COUNTING BY COMPUTER IS EASY."
- 3) Write a program which will assign a value to each of two variables. It will then tell the user if the numbers are equal and if not, which is the larger number.

Chapter VIII

INPUT:

TALKING BACK TO THE COMPUTER (BASIC)

Glossary:

INPUT A statement which allows data to be entered
Statement from the keyboard into the program without
----- changing the BASIC program.

In the programs we have discussed up to this point, the computer has done all of the work while a program ran. An INPUT statement, though, causes the computer to stop part way through the program. It waits while the user types some data into the machine. When the user has finished, the computer then continues to execute the program. In addition to stopping the program, the INPUT statement also acts in a manner similar to that of a LET statement. The computer sees the INPUT statement as an assignment to a variable. While the computer waits for the user to type in data, it is, in essence, waiting for a value to be assigned to the variable.

When the computer executes an INPUT statement, it PRINTs or displays a question mark. The question mark acts as a cue or a prompt. The purpose of this is to remind the user that it is his/her turn to do something.

Software products almost always have INPUT statements. These allow users to balance checkbooks, do financial planning or play games by changing the numbers upon which the program acts. Again, the important thing to remember is that nothing will happen after the execution of an INPUT statement until the computer receives an acceptable answer.

Instruction:

You type:

```
10 PRINT "PICK A NUMBER"
20 INPUT A
30 PRINT "PICK A NUMBER"
40 INPUT B
50 PRINT A+B,A-B,A*B,A/B
```

Let's analyze this program:

Line 10 is a PRINT statement which tells the user what he/she should type into the computer.

Line 20 is an INPUT statement which forces the computer to display a question mark and then wait until the user provides some data. When the user types a number (followed, of course, by {RETURN}), BASIC places the value of the number into the variable named A.

Line 30 is a repetition of line 10.

Line 40 is a repetition of line 20 except that the variable used is named B.

Line 50 is a PRINT statement which tells the computer to display the sum, difference, product and quotient of the two numbers.

```
-----
| NOTE: In the description above, |
| we use the word "user" to mean |
| the person who is RUNning the  |
| program. As the programmer, you |
| are or can be distinct from the |
| "user". Try it! Let someone    |
| else RUN this program. The      |
| difference between this and pre- |
| vious programs is that in this  |
| example the program interacts    |
| dynamically with the user.      |
|                                  |
|-----|
```

Instruction:

You type:

RUN

Try this program several times with different answers. Also, try purposely hitting just {RETURN} in response to the prompts. Did you find any problems? Notice how the computer must stop each time it reaches an INPUT statement. Another way to write the program would be.

Instruction:

You type:

NEW

```
10 PRINT "PICK TWO NUMBERS"
20 INPUT A, B
30 PRINT A+B, A-B, A*B, A/B
RUN
```

When you RUN this program, the computer will display the PRINT statement. On the next line, a question mark will be shown. Nothing will happen until the user enters two numbers into the computer. This can be accomplished either by typing two numbers separated by a comma or by typing one number and {RETURN}, and another number and {RETURN}. The computer, in the latter case, will not continue with the program until two numbers have been entered. Also the computer will display another question mark to remind the user that it is waiting for another number.

There are some problems that you might encounter. If the user enters two numbers without a comma to separate them, the computer will display "ERROR 8 INPUT". This simply means the computer has received inappropriate data. Remember, computers are not as smart as people. Information entered may not vary from the prescribed format. A missing comma may be no big deal to a person; but, to the computer, it is a huge deal.

Also the computer will not allow the user to put a space before the comma. Spaces before each number are permitted but not before the comma.

Acceptable

25,5
25, 5

Unacceptable

25 5
25 , 5

For these and other reasons, it is generally advisable to use only one variable per INPUT statement.

INPUT statements are often used in educational programs. By using INPUT statements one could test a student's knowledge of mathematics. For example, if you wanted to test a student's ability to determine the area of a rectangle, you might use a program like this:

Instruction:

You type:

```
10 PRINT "WHAT IS THE LENGTH"
20 INPUT L
30 PRINT "WHAT IS THE WIDTH"
40 INPUT W
50 PRINT "WHAT IS THE AREA"
60 INPUT A
70 IF A = L*W THEN PRINT "THAT'S CORRECT"
80 IF A <> L*W THEN PRINT "SORRY, THAT'S NOT RIGHT"
```

If the student knows the formula for calculating the area of a rectangle, this small program also will provide an adequate test of his/her multiplication skills.

Exercises:

- 1) Write a program that will compute the area of a triangle. The formula for determining the area of a triangle is $\text{BASE} \times \text{HEIGHT} / 2$.
- 2) Write a program that will allow 3 test scores to be entered and which will then PRINT the average of those scores.
- 3) Write a program which will display all whole numbers and their squares, starting at one and ending at a number determined by a user's INPUT.

Chapter IX

LOGICAL OPERATORS: DOES THIS MAKE SENSE

Glossary:

Logical Operators

AND Is used to connect two conditions and to
--- determine if both conditions are met.

OR Is used to connect alternative conditions
-- and determine if either one condition or
 the other is met.

In addition to the relational operators, discussed in Chapter VII, BASIC also uses logical operators. These operators include AND and OR. The logical operators allow the computer to make more complex decisions than were possible with just the relational operators. Almost always logical operators are used in conjunction with relational operators.

Examples:

```
60 IF X=9 AND Y>15 THEN PRINT X*Y
90 IF MONTH<12 OR DATE>26 THEN GOTO 10
```

Although AND and OR are similar, the computer reacts to them in very different ways.

Using the AND operator, the chart below shows that the only time a statement is true is if both conditions are true:

```
True AND True = True
True AND False = False
False AND True = False
False AND False = False
```

You read this chart to mean (for example) "When the first comparison is TRUE AND the second comparison is TRUE the entire condition is TRUE." On the other hand, the OR chart produces more true statements:

True OR True = True

True OR False = True

False OR True = True

False OR False = False

For a real life example using logical operators such as AND and OR, imagine that you and a friend are shopping for tonight's dessert. You will buy ice cream for dessert if AND only if both you AND your friend choose ice cream. On the other hand, if your allowances are a bit more generous, perhaps you could buy ice cream if either you OR your friend choose ice cream for dessert.

With the logical operator AND, you both must want the ice cream. Two conditions must be true. Using the logical operator OR, only one condition must be true -- either you OR your friend must want ice cream.

In a computer statement using the logical operator AND, if any part of the statement is false, the whole statement is false. The logical operator AND forces the computer to make a decision.

Remember, when using the IF statement, if the statement is true, the computer will execute one set of instructions (those following the THEN). If the statement is false, the computer will skip those instructions and will continue to execute the instructions on the next line.

Examples:

```
IF 4<5 AND 10>6 THEN PRINT "BOOK"
```

Whenever the computer reaches the above line, it will ascertain that 4 is less than 5 AND 10 is greater than 6. Since both relational conditions of the IF statement are true, the computer will PRINT the string "BOOK".

```
IF X=C OR D=T THEN PRINT X-T
```

In the above example, the computer must decide if X is equal to C. It must also determine if D is equal to T. If either is true, the computer will display the difference between X and T. One relational condition (X=C) or the other relational condition (D=T) must be true. If both are true, the computer will display the difference between X and T. If both relational conditions are false, the computer will simply drop down to the next consecutive program line.

IF 80<100 AND Z<A THEN GOTO 40

Again, the computer will ascertain if each relational condition is true. If both and only if both are true, the computer will proceed to line 40. If either relational condition is not true, the computer will proceed to the next consecutive program line. (Note that, since 80 is always less than 100, this conditional test actually depends only on the truth of Z<A.)

IF 100=G OR 15>5 THEN GOTO 50

In the above example, the computer will decide if 100 is equal to the value of the variable G. If it is or if 15 is greater than 5, the computer will proceed to line 50. If both relational conditions are false, the computer will drop down to the next consecutive program line. (Again, since 15 is always > 5, the conditional is always TRUE, so the value of G really doesn't matter.)

NOTE: Since you, as the programmer, can always know the truth value of comparisons of constants, you would seldom (if ever) put such comparisons in a program. They are included here for illustration purposes only.

Instruction:

You type:

```
10 PRINT "WHAT WAS YESTERDAY'S TEMPERATURE";
20 INPUT YESTEMP
30 PRINT "WHAT IS TODAY'S TEMP";
40 INPUT TODTEMP
50 IF YESTEMP > 80 AND TODTEMP > 80 THEN PRINT "WE HAVE
    A HEAT WAVE"
```

There are some important concepts in the above program. First, notice the semicolon following our PRINT statements. Remember a semicolon suppresses the return to the new line and an INPUT statement automatically produces a question mark. By using the semicolon in our PRINT statement prior to the INPUT statement, the question mark directly follows our question. This is no big deal. If you don't use the semicolon, it will not alter the function of the program. However, the semicolon does make the screen look nicer when the program runs.

In line 50 we have used our logical operator AND. Notice the computer is forced into a decision. If the computer ascertains that the statement is true, it continues to execute the portion of the line following THEN. On the other hand, if the computer finds that the statement is false, it attempts to execute the next line. Since there is no following line, the program ends.

RUN this program and answer the question to see what happens. Try it several times with different temperatures.

Instruction:

You type:

50 IF YESTEMP < 60 OR TODTEMP < 60 THEN PRINT
"MY, YOU SURE HAVE SOME COLD DAYS".

(Remember, we have changed line 50 by just typing 50 and the new line. The computer will automatically disregard the previous line 50).

Again, we have forced the computer to make a decision. If either condition in line 50 is true, the computer will continue to execute the line. If neither is true, the computer stops.

Again, RUN the program several times, giving various answers to the questions.

Exercises:

-
- 1) Write a program which will compare the scores of two bowling games for you and your opponent.
 - 2) Write a program which will PRINT a message if a number INPUT by a user is between 10 and 20.
 - 3) Write a program which allows the user to INPUT three numbers. Display the numbers only if they were typed by the user in numerical order, from smallest to largest.

Chapter X

RANDOM:
I WON WHAT?

Glossary:

Random -----	The process by which an item is chosen Selection without a pattern or a definite aim.
RANDOM -----	A function which asks the computer to automatically select a random number from between specified boundaries.
Integer -----	A whole number; a number with no fractional or decimal part.

You have probably witnessed or taken part in a random drawing. People buy tickets to win a prize. One part of the ticket is kept by the buyer; the other part is thrown in a hopper or a hat. One winning ticket is selected without definite aim. All tickets have an equal chance to be chosen. This is a random selection.

Another type of random selection involves board games. These games usually come with a spinner, die, or dice. Although the number of choices is limited, the actual number is selected by chance. Each number has an equal opportunity to be selected.

Using BASIC, a computer is capable of making random selections. Because computers are often required to select a random number, the function for choosing a random number is built into the computer language. You could write your own program that would select a number at random, but it is easier to use the random function which is built into the language.

In BASIC XL, the random function is written RANDOM(,). The blank spaces indicate numbers to be supplied by the programmer. The first number determines the lower limit of the range of numbers to be selected. The second number determines the upper limit of the range of numbers to be selected. The two numbers are separated by a comma. The RANDOM function always selects integers (whole numbers).

Examples:

RANDOM(1,6) selects integers between 1 and 6 inclusively. This is identical to throwing a single die in a dice game.

RANDOM(1,100) selects integers between 1 and 100 inclusively.

RANDOM(5,50) selects integers between 5 and 50 inclusively.

RANDOM(100,1000) selects integers between 100 and 1000 inclusively.

```
-----
| Since BASIC XL considers the left |
| parenthesis to be part of the name of |
| the function 'RANDOM(', there can be no |
| space after the 'M' when the name is |
| typed in. This rule holds true for all |
| BASIC XL functions. |
-----
```

There is one major difference in the way the computer chooses a random number and our random prize drawing example. In the random prize drawing, if more than one number is to be selected, the first number drawn is put aside. The remaining numbers now have a better chance of being selected. When the computer selects a random number, it tosses the number back into the "hopper". Each time a number is selected, it has the same probability of being selected the next time. If, for example, you wanted the computer to choose 15 numbers between 1 and 100, it is possible that one or more numbers might be repeated. It is not an easy task to keep the computer from reselecting a number, and the topic will not be discussed here.

Selecting a random number is the basis for a simple guessing game. As you learn more about computer programming, you will be able to use variations of this game to form a more sophisticated program.

Also please note: to select a number with zero as the lower limit, it is not necessary to write the zero in the RANDOM function. When only one number is present, the computer assumes the lower limit to be zero. However, it then assumes that the upper limit is ONE LESS than the number given. (There is a historical reason for this: Apple II Integer BASIC functions this way.)

RANDOM(100) selects integers between 0 and '99 inclusively.

RANDOM(50) selects integers between 0 and 49 inclusively. The RANDOM function tells the computer the range of numbers from which it is to make its selection. You must tell the computer how many numbers to select.

Instruction:

You type:

```
10 LET I = 1
20 PRINT RANDOM(1,100)
30 LET I = I + 1
40 IF I < 7 THEN GOTO 20
```

The above program will select 6 random numbers between 1 and 100. Let's analyze each statement line. First, we initialized our variable I in line 10. In line 20 we tell the computer to PRINT a random number. Next, we use our non-algebraic equation as a counter. Each time the computer executes line 30, the value of I increases by one. Finally, we force the computer to make a decision. The computer must decide if the value of I is less than seven. If it is, the computer returns to line 20 and executes that line and the succeeding lines. On the other hand, if the computer determines that I is equal or greater than 7, it stops.

Instruction:

You type:

```
10 LET NUMBER = RANDOM(1,100)
20 PRINT "PICK A NUMBER BETWEEN 1 and 100"
30 INPUT GUESS
40 PRINT GUESS , NUMBER
```

The above program represents a game where the computer selects a number, and the user tries to guess the number. Notice the value of NUMBER is chosen at random by the computer. The number 1 is the lower limit and 100 is the upper limit from which the computer might make its selection. In line 20, the PRINT statement indicates which kind of data the user should enter into the program. The INPUT statement makes the computer wait until the value of the numeric variable has been entered. Notice GUESS is a numeric variable. The user may choose any number between 1 and 100. Thus, any value between 1 and 100 may be assigned to our variable GUESS. In line 40, the PRINT statement tells the computer to display the GUESS and the NUMBER it chose. The comma suppresses the carriage return so that both the number and GUESS are displayed on the same line. Again, this line shows the two variables: the NUMBER variable whose value was determined by the computer and the GUESS variable whose value was determined by the user.

Exercises:

1) Write a program to generate 10 random numbers in the range 1 to 50.

2) Modify the above program so that the range of numbers is 0 to 999.

3) Modify our guessing game example so that the computer picks a random number between 1 and 50, and you try to guess the number. Remember to use INPUT.

4) Modify the above program so that the computer PRINTs hints to the user to trap the number. Hints might include "TOO SMALL" and "TOO LARGE". Allow the user to keep on guessing until he gets the correct answer.

CHAPTER XI

THE PROGRAM RECORDER: HITS ON TAPE

The following is an explanation of how to use the Atari program recorder. If you own one of these devices, this chapter will explain how to preserve your programs on tape and how to retrieve them. If you do not own a recorder, the information that follows should enhance your understanding of the product and also help you to decide whether or not to purchase a program recorder.

GLOSSARY:

Program Recorder: -----	Is a cassette tape recorder that can be used to transfer programs to and from the Atari Home Computer.
CLOAD -----	A command which is used to enter a program from the program recorder to the computer's memory and is used in conjunction with CSAVE.
CSAVE -----	Is a command which is used to store a computer program from the computer's memory to the program recorder and is used in conjunction with CLOAD.
LIST "C:" -----	Is a command which directs the computer to store programs or specified program lines on a cassette tape and is used in conjunction with ENTER "C:".
ENTER "C:" -----	Is a command which directs the computer to enter a program from the program recorder to the computer's memory. It is always used in conjunction with LIST "C:".

By now you are getting some ideas about computer programming. Perhaps you are beginning to see how this new skill will make life simpler, or perhaps you are

becoming aware of how the computer can be used in your home, office, or school.

As discussed earlier, once you turn off your computer, all the programs you were working on are erased from the computer's memory. This is also true if you type "NEW" into the computer (or open the cartridge door on the Atari 400 or 800). However, there are times when you want to save a program. You could use paper and pencil and write down your program. Then, each time you want to use it, you would have to re-enter the program into the computer's memory. This is very time consuming and also very frustrating, especially if it is a program you often use.

Fortunately, there is an easy way to save your programs. You can purchase an ATARI Program Recorder. This recorder is very similar to a portable tape recorder, but it is designed specifically to work with an ATARI computer. A 30 minute cassette will conveniently hold two of your computer programs, one on each side. Also, if you own a program recorder, you can buy many program tapes. These tapes contain various computer programs that will help with family budgeting, learning a foreign language, or games, just to name a few.

Be careful when handling the cassettes. They are easily damaged, particularly if you touch the tape itself. Be sure to store the cassettes in their cases when not in use. Do not store the tapes in hot areas, direct sunlight, or near magnetic fields, such as those found near motors, magnets, or airport security detectors.

It is a good idea to label every cassette with the names of the programs it contains. This will make it easier for you to locate a particular program when you want it. Notice, each cassette has two notches in the rear edge. The Program Recorder will not record on a cassette tape which has the holes exposed. After recording one program or two programs on a tape, you can protect the program(s) from being erased or taped over by punching out the square of plastic and exposing the holes.

That should be enough background information to get you thinking about purchasing a Program Recorder. Now the important stuff. You've just finished writing a brilliant program, and you want to save it. You may type one of two statements in order to save your program. One statement is CSAVE; this can only be used for storing programs on a cassette. The other statement is LIST"C:". (A variation of this statement can be used with other devices in addition to the program recorder.)

CSAVE will always save the entire program from the computer's memory. With LIST"C:", you have two choices. You can save the entire program or, if you wish, you can specify the first through last lines to be saved. For example,

```
LIST "C:",200,1500
```

would cause all program lines between 200 and 1500 to be listed on a cassette.

Whichever statement you use, CSAVE or LIST"C:", each causes the computer to react in the following manner. First, the computer will beep twice. This is your signal to put the cassette into the program recorder and to move the tape using "Rewind" or "Fast Forward" to the point at which you want the recording to begin. Press the {Record} and {Play} levers. Since the computer cannot determine when you are finished setting up the tape, you must signal it by pressing the {RETURN} key. Once you do that, the tape will start moving. If you turn up the volume on your monitor, you will hear the recording taking place. When the recording is completed, the tape will stop moving and you can press the {Stop} lever.

If you have saved your program on cassette by using a CSAVE statement, then you must use the CLOAD statement to get your program from cassette into the computer's memory. If you used LIST"C:" to store your program on cassette, then you must use ENTER"C:" to load programs back into the computer.

CLOAD will erase the program currently in memory before loading a new one. On the other hand, the ENTER"C:" statement will merge the program it loads with the program in memory. However, if the incoming line numbers are the same as the existing ones, the incoming lines will replace the existing ones. To avoid the merging, type "NEW" before using the "ENTER" statement.

When you use CLOAD or ENTER"C", the computer will beep once. Again, this is your signal to get the cassette into the program recorder and to adjust the tape to the point at which the program begins. Press the {Play} lever on the program recorder and then the {RETURN} key on the computer's keyboard. The tape will begin moving. When the tape stops, press the program recorder's {Stop} lever.

After your program is loaded into the computer's memory, simply type RUN and press the {RETURN} key. This executes your program.

In order to see one advantage of using the LIST"C:", try this exercise:

Instruction:

You type:

100 PRINT "THIS IS LINE 100."
110 PRINT "IN ORIGINAL PROGRAM"
200 PRINT "THIS IS LINE 200"
210 PRINT "IN ORIGINAL PROGRAM"
300 PRINT "THIS IS LINE 300"
310 PRINT "IN ORIGINAL PROGRAM"

Now RUN this program to see what it does.

Then type:

LIST"C:",200,210

When you hear the two beep signal, place the cassette into the Program Recorder. Push the Play and Record lever. Press the {RETURN} key. When the tape stops moving, press the {Stop} lever on the Program Recorder.

Now type:

NEW

100 PRINT "THIS IS A NEW LINE 100"
200 PRINT "THIS IS A NEW LINE 200"
300 PRINT "THIS IS A NEW LINE 300"

Now RUN the program to see what it does. Rewind the cassette tape and then type "ENTER"C:". After the beep, press the {Play} lever on the recorder and the {RETURN} key on the keyboard. When the tape stops, press the {Stop} lever.

To see the results, type LIST and RUN. Notice how the programs have merged. This may not seem important to you now, but as your programming ability increases, this will become a handy procedure. It allows you to alter parts of programs or to combine programs easily. Again, this can save you valuable time and expand your programming horizons.

Exercise:

1) Practice using the program recorder by copying any of the programs previously presented in this book.

Chapter XII

THE DISK DRIVE: BEING FLOPPY ISN'T SLOPPY

The information contained in this chapter is intended to give the reader general information on disk drives and their uses. If you own one of these devices, this chapter should enhance your knowledge of how to use the drive. If you do not own one of these devices, the information in this chapter will help you determine if you should or should not purchase this additional piece of equipment.

Glossary:

Disk Drive -----	Is a device which connects to the Atari Home Computer and reads or writes information on a diskette.
Diskette -----	Is a vinyl "record" enclosed in a stiff plastic envelope.
File Name -----	Consists of up to eight characters and is used to distinguish files on a diskette.
File Name Extension -----	Is a suffix or an addition to a file name and consists of a period and one, two, or three characters.
Boot ----	Is the method for starting the disk drive and loading its operating system into the computer's memory.
DOS ---	An acronym for Disk Operating System. Also, a command from BASIC XL which will transfer control to DOS (not discussed in this book).
System Diskette -----	Any diskette which contains a DISK OPERATING SYSTEM (DOS) on the diskette.

DIR Is a command which causes the computer to list all the files currently located on a particular diskette.

LOAD Is a command which causes a program previously stored on a diskette to be entered into the computer's memory. This command is used in conjunction with SAVE "D:FILENAME.FNE".

SAVE Is a command which causes a program in the computer's memory to be stored on a diskette. This command is used in conjunction with LOAD "D:FILENAME.FNE".

ENTER Is a command which causes a program previously stored on a diskette to be entered into the computer's memory. This command is used in conjunction with LIST "D:FILENAME.FNE".

LIST Is a command which directs the computer to store programs or specified program lines on a diskette. This command is used in conjunction with ENTER "D:FILENAME.FNE".

In the previous chapter, we discussed the advantage of owning an Atari Program Recorder. If you enjoy programming or if you are planning on using a great deal of the commercially produced software programs, you should consider purchasing a disk drive. Although it initially costs more than the program recorder, the disk drive substantially increases the usefulness of your Atari Home Computer.

A disk drive is more effective as a program storage device than a program recorder. A single diskette could hold as many as 64 programs while a cassette conveniently holds only two program. It operates quicker and permits almost instantaneous access to information. The disk drive also is more reliable. With a disk drive, you have more choices of prepared software. Also, the diskette used with the disk drive keeps track of all the information stored on it. Unlike the program recorder, you do not have to remember where a program is stored. The diskette (or, more properly, the Disk Operating System) remembers for you. There are other major advantages to disk drives versus cassettes, particularly as the programs written or used become more sophisticated.

Before we get ahead of ourselves, let's begin by explaining the physical aspects of the disk drive.

The disk drive is a rectangular box which connects directly into the Atari Home Computer. (In the Atari 1450XLD a disk drive is built into the system.) Imagine a record player which only operates when the lid is shut. Essentially that is how the disk drive works. While a record player uses a needle to produce sound from a record, the disk drive uses a magnetic head that can read or write information on a special "vinyl record" called a diskette.

If you examine a diskette carefully, you will notice that there are openings in the plastic envelope surrounding the "vinyl record". These openings allow the disk drive to read information from or write information on the diskette without having to remove the protective plastic cover.

On the upper right side of the diskette, there is a notch. This notch works in the same way as the plastic squares on the rear edge of the cassette tape. If the notch is present, then information can be recorded on the diskette. If it is not present, the disk drive will not write on that particular diskette. Some diskettes contain no notch; thus, they are permanently protected from accidental writing. If you store information on a diskette and you want to be sure it is protected, you can cover the notch with a special label called a "write protect tab" or with a piece of opaque tape.

Although a diskette is small, 5 1/4 inches in diameter, it can store a great deal of information. A diskette is often compared to a file drawer. The diskette is capable of holding many files in the same way that a file drawer can hold many file folders.

In order to distinguish each file from the other files stored on a diskette, a name is assigned to each file. A name consists of a maximum of eight characters. The first character must be a capital letter. However, the name itself can contain a combination of capital letters or capital letters and numbers. No blank spaces, punctuation marks, or special characters are permitted in file names.

To further distinguish one file from another, you might include a file name extension. A file name extension consists of one, two, or three characters and may contain any combinations of letters and numbers. To specify a file name extension, simply add a period to

the end of the file name and then add the extension. For example, BUDGET.88 might be the name of your family budget program for 1988.

A reserved section of the disk, called the directory, remembers these names and file name extensions for you (along with their locations and sizes). When you use the name of a file that you think is already on the disk (as in LOAD or ENTER), the directory for your file and "connects" BASIC and your program with it if it exists. If it doesn't exist, DOS indicates an error. When you use the name of a new file (as in SAVE), DOS creates an entry in the directory for you.

NOTE: If you have Atari DOS 2.0, DOS XL version 2, or OS/A+ version 2, you may ignore this note. If you have such a DOS, you should refer to its reference manual for information regarding acceptable file names. Some Disk Operating Systems currently available for the Atari allow longer and more complex filenames. OS/A+ version 4.1, for example, allows 30-character names and allows almost any character in a name.

In order to use a disk drive, be sure that it is properly connected to your Atari Home Computer. If you have questions about this, check your Disk Drive Reference Manual.

Next, if your computer is on, turn it off. Turn on the disk drive and insert a system diskette. Follow the procedure for properly inserting the diskette according to your disk drive manual. Be sure that your fingers touch only the protective, plastic envelope. The diskette should slide into the drive. Gently close the disk drive door. If there is any resistance while inserting the diskette or closing the disk drive door, stop, remove the diskette, and try again. This is an easy procedure and does not require brute strength. If you use force, you could ruin the diskette. Turn the computer on; the disk drive will make a whirring noise. This means the disk operating system is loading from the diskette into the computer's memory. This procedure is known as "booting". When the READY message appears, the boot procedure is completed.

If you do not use the proper diskette, a BOOT ERROR will result. A proper diskette is one that contains a disk operating system, often referred to as DOS.

```
-----  
| Some DOS disks (such as OS/A+ version |  
| 2.1) do not automatically enter the |  
| cartridge when the power is turned |  
| on. The user must type some command |  
| ('CAR' in the case of OS/A+) to enter |  
| the BASIC XL cartridge.              |  
-----
```

Once you have successfully booted a diskette, you will want to know what programs are located on it. To get the "directory" or listing of the files, type DIR {RETURN} into the computer. This will list the diskette directory on the screen.

If you have digested all of the above information, you are now ready to use your diskette. Choose the program, from the directory that you would like to run. Type LOAD "<name.ext>". When the computer has loaded the program from the diskette to its memory, it will say READY. Next, type RUN. The computer will execute the program.

If you have created a program and you would like to store it on your diskette, first choose one with DOS (DISK OPERATING SYSTEM) already on it. Then, type SAVE "D:<name.ext>". This will store an entire program from the computer's memory onto the diskette.

Remember, if you use the command SAVE to store a program on a diskette, then you must use the command LOAD to retrieve the program from the diskette and record it in the computer's memory. SAVE and LOAD are used to store and retrieve for an entire program.

If you want to keep part of a program, then you may use the command LIST"D:<name.ext>",<line range>. This command also allows you to preserve a particular program line or range of files. For example:

```
LIST"D:INCOME.87",310
```

will record on a diskette only line 310 (if it exists) of the program currently in memory. It gives the filename "INCOME.87" to the file containing that single line.

If, you want to keep only lines 310 to 450, you would use the following command:

```
LIST"D:INCOME.87",310,450
```

This command will preserve only lines 310 through 450. If there are additional lines in the program, they will not be preserved on the diskette. To keep the entire program, simply use LIST"D:INCOME.87" without any line numbers.

Finally, the commands LIST and ENTER must be used together. One advantage of using LIST and ENTER over SAVE and LOAD is that the LIST and ENTER method allow you to merge programs. LOAD will erase the program currently in the computer's memory before loading a new one. On the other hand, the ENTER"D:FILENAME.FNE" statement will merge the program it loads with the program in memory. As with keyboard programming, if the incoming line numbers are the same as the existing ones, the incoming lines will replace the existing ones.

In order to see the advantage of using LIST "D:FILENAME.FNE", line, try this exercise:

Type:

```
100 PRINT "THIS IS LINE 100"
110 PRINT "IN ORIGINAL PROGRAM"
200 PRINT "THIS IS LINE 200"
210 PRINT "IN ORIGINAL PROGRAM"
300 PRINT "THIS IS LINE 300"
310 PRINT "IN ORIGINAL PROGRAM"
```

Now RUN this program to see what it does. Then, type LIST "D:SILLY.PRG", 200,210.

Now Type:

```
NEW
100 PRINT "THIS IS A NEW LINE 100"
200 PRINT "THIS IS A NEW LINE 200"
300 PRINT "THIS IS A NEW LINE 300"
```

Now RUN this program to see what it does.

Type:

ENTER "D:SILLY.PRG"

To see the results, type LIST and RUN. Notice how the programs have merged. This may not seem important to you now. but as your programming ability increases, this will become a handy procedure. It allows you to alter parts of programs or to easily combine programs. Again, this can save you valuable time.

COMMENTARY: One use for the file name extension is as an indicator of which command was used to store a program. Once you see the program name in the directory, you will know which command to use to retrieve the program into the computers memory.

Here is a sample directory:

```
*  DOS          SYS 004
*  INIT         COM 006
*  RS232        COM 001
*  COPY         COM 075
    DEMO        LIS 014
    NEWDEMO     LIS 007
    MOVE1       SAV 004
*  MOVE2        SAV 004
592 FREE SECTORS
```

The first file shown is DOS.SYS or the Disk Operating System. The next several files are for utility programs which were previously stored on the diskette and permits access to the diskette in different ways. Also, several programs are preceded by an asterisk. The asterisk indicates that that program is protected and cannot be renamed or erased or written to until it is unprotected. If you desire more information, please consult the OS/A+ or DOS XL Reference Manual.

The other programs stored on the diskette were those used as examples in this book. Notice the file name extensions. Some programs use LIS for programs which had been LISTed to the diskette, and others use SAV for programs previously SAVED to the diskette.

In the process of placing a program in the computer's memory from the diskette, it is important to follow the correct procedure. That is, if the user employs ENTER with a program previously stored with SAVE or LOAD with a program previously with LIST, an ERROR will be the result.

Exercises:

- 1) Type in a program. SAVE it to a disk file with a name of your choice. Type NEW and then LOAD your program and RUN it. Did it work?
- 2) Try to LOAD a LISTed file. What ERROR did you get?
- 3) Try to ENTER a SAVED file. What ERROR did you get?

CHAPTER XIII

THE PRINTERS: HARDCOPY ISN'T HARD

As with the previous two chapters, this chapter is intended to give the reader some general information concerning printers. If you already own one of these devices, the information which follows will enhance your understanding. If you do not own one of these devices, the information contained in this chapter should help you determine whether or not to purchase a printer.

Glossary:

Software -----	The programs or instructions which make the computer perform specified tasks.
Hardware -----	The equipment which makes up a computer system. Hardware includes the computer, monitor, disk drive, program recorder and printer.
Printer -----	A device which produces on paper characters previously stored in the computer's memory. Printers are also known as line printers.
Hardcopy -----	Is the same as a paper copy.
LIST"P:" -----	A statement used for obtaining a hardcopy listing of a program previously stored in the computers memory.
LPRINT -----	A statement which causes the computer to produce output on paper.
Output -----	Many programs utilize BASIC statements which cause information to appear on the screen or printer. Such information is referred to as output because the computer puts the information out on the screen or on the printer.

Throughout this book we have made a concerted effort in our explanations not to use jargon or computerese. Perhaps a few buzz words won't hurt now.

Often computer discussions will include the terms "software" and "hardware". Software refers to computer programs. If you buy a computer game or any program, you are buying software. Hardware, on the other hand, is the equipment and includes the computer itself, the monitor or television, the disk drive, program recorder, and the printer.

A printer is a device which produces, on paper, characters previously stored in the computer's memory. Printers are manufactured by ATARI and many other companies. Some are designed to operate with your Atari Home Computer. Before you purchase a printer, check with the dealer to be certain that the printer will work with your computer.

Why do you need a printer? Truthfully, you may not. A printer produces a paper copy of your program or the information produced by your program. A paper copy in computer jargon is called a hardcopy. What you see on your monitor or television is softcopy. Hardcopy can be held in your hand; softcopy cannot be held.

Hardcopy is desirable if you want to produce business or financial reports. Also, hardcopy is essential if you use your computer to produce letters. For a programmer, hardcopy provides a means for reviewing a program whose length exceeds one screen size.

Printers come in a variety of sizes. The amount you pay for a printer will be determined by its speed and the quality of the print. Typewriter quality print is the most expensive kind of print. Other printers are capable of producing graphic displays. Some are even able to reproduce color.

If you decide to buy a printer, spend some time looking at the various models available. Then determine the features you want and make your selection.

In order to use a printer there are two commands you need to know. LIST"P:" will cause the computer to PRINT your entire program. If you want only one line to appear on paper add the line number to the LIST"P:" statement. Alternatively, if you include two numbers separated by a comma following the LIST"P:" statement, the computer will PRINT those lines inclusively.

Examples:

LIST"P:" Will list your entire program.

LIST"P:",15 Will list only line 15.

LIST"P:",30,56 Will list lines 30 through 56
inclusively.

The other command used in conjunction with a printer is LPRINT. LPRINT (Line PRINT) causes the computer to PRINT data or output on paper rather than on the screen. If you want to see results on paper instead of the screen use LPRINT.

Example:

```
10 LET P=1
20 LET X = RANDOM(1,100)
30 PRINT X
40 LET P=P+1
50 IF P<11 THEN GOTO 20
```

The above program will produce 10 random numbers between 1 and 100 on your monitor screen. By changing line 30 to:

```
30 LPRINT X
```

the same results will be produced on paper.

You must have a printer and you must turn on your printer in order to use the LPRINT and LIST"P:" statements. If you try to use these statements without a printer, an ERROR 138 will be the usual result.

```
-----
| The command LPRINT forces the computer to |
| the beginning of a new line. Using ; or , |
| as we did with PRINT on the end of a line |
| may not produce the results you desire. If |
| you need to produce more complex printed |
| output, you may have to avoid LPRINT.    |
| Please refer to the OPEN and PRINT sections |
| of your BASIC XL Reference Manual.        |
|-----
```

Exercises:

- 1) Use LPRINT statements to write on your printer a note to a friend.
- 2) List your program to the printer.

CHAPTER XIV

GRAPHICS PART I: I GET THE PICTURE

Glossary:

Graphics -----	A state in which the computer responds to Mode instructions for the purposes of drawing pictures, designs, graphs, or variations of the standard characters.
Pixels -----	Shaded blocks of colors used in GRAPHICS modes to create pictures, designs and graphs.
Graphics Window -----	The large area of the monitor or television screen in which graphics, words, designs, or pictures can be displayed.
Text Window -----	An area at the bottom of the monitor or television screen which contains enough space for four lines of text.
COLOR -----	A statement which selects one of the available colors which will be used with subsequent PLOT and DRAWTO statements.
PLOT -----	A statement which illuminates a single point on the screen.
DRAWTO -----	A statement which causes a line to be drawn from a point (the last plotted point) to a specified location.

One of the most exciting features of your ATARI Home Computer is its graphics capabilities. Using the various graphics modes can enhance any program you write. Your ATARI is capable of being used in many different graphic modes. Some graphics require more than a beginner's understanding of programming and computer design. This chapter is intended to give you an introduction into the world of graphics and for that reason we will be discussing only modes 0 and 7, the most commonly used modes.

To enter the graphics mode, type GRAPHICS and the number of the mode you desire. In this chapter, as noted, we will only use GRAPHICS 0 and GRAPHICS 7.

To better conceptualize the graphics modes, think of your screen as a piece of graph paper. In mode 0, the screen is equal to 40 columns X 24 rows. In mode 7 the full screen is equal to 160 rows X 96 columns. (Rows run horizontally and columns run vertically.)

GRAPHICS 0 is the mode we have been using throughout this book. Whenever you turn on your ATARI, the screen mode is automatically set to GRAPHICS 0. In some ways, it might be easier to think of GRAPHICS 0 as turning OFF graphics. In GRAPHICS 0 you cannot PLOT and DRAWTO as you can in other modes.

Also, in GRAPHICS 0 what appears on the screen are characters. The numbers, letters and special symbols that are on the ATARI keyboard are also displayed on the screen.

In graphics mode 7, you have more choices. First, there are more columns and rows than were found in mode 0. Unlike GRAPHICS 0, though, GRAPHICS 7 uses pixels instead of characters. Pixels are shaded blocks of color. Remember our comparison of the monitor screen to a piece of graph paper. A pixel represents one square which is or can be filled with color.

Instruction:

You type:

GRAPHICS 7

What changes took place? Notice the physical changes in the screen. First, the screen is split. There is a graphics window which is black and a text window which is blue. (If you have a black and white monitor or TV, obviously you will only see shades of gray. In most cases you will be able to distinguish the various colors anyway.) The graphics window is the large area of the monitor or television screen in which graphics words, designs or pictures can be displayed. The text window is the area at the bottom of the monitor or television screen which contains enough room for four lines of text or program statements to be displayed.

The pictures or designs you create will appear in the larger, graphics screen, while your program statements, prompts, messages, etc., appear in the text window. Note that displays in the text window appear to be the same as those in GRAPHICS 0. This is proper, since the text window is actually and simply a small GRAPHICS 0 screen.

Also in mode 7, you have a choice of colors. The COLOR statement allows you to select one of the available colors and to draw with that color. In graphics mode 7, there are four available colors: black, orange, green, and blue. The black is used for background; but, along with the other colors, it may also be used by PLOT and DRAWTO.

The PLOT statement enables you to tell the computer a particular point position you desire. Again, think of your screen as a graph. The first number after PLOT tells the computer the column desired; the second number indicates the desired row. The screen positions are numbered from the upper-left corner of the screen starting with the number zero (0). Numbers in PLOT statement are separated by a comma.

		columns									
		0	1	2	3	4	5	6	7	8	9
	0	X					X				
	1										
	2			X							
r	3										
o	4										
w	5	X									
s											

The points shown in the above examples would be entered into the computer as follows:

```

GRAPHICS 7
COLOR 1
PLOT 0,0
PLOT 5,0
PLOT 2,2
PLOT 0,5

```

Another statement used in most graphic modes is the DRAWTO. The DRAWTO statement tells the computer the position at which to END a line. That means, the PLOT statement tells the computer to start a line at the position given, and the DRAWTO statement tells the computer to end the line at the position given. Just like the PLOT statement, the DRAWTO statement also uses two numbers and a comma.

PLOT 0,0
DRAWTO 15,25

Here are some examples for you to try.

Instruction:

You type:

{SYSTEM RESET}
GRAPHICS 7
COLOR 1
PLOT 5,5
DRAWTO 96,64
PLOT 150,0
DRAWTO 96,64

What did you see? You should see a large, somewhat misshapen "V" drawn in orange.

{SYSTEM RESET}
GRAPHICS 7
COLOR 1
PLOT 10, 10
DRAWTO 159, 10
COLOR 2
PLOT 10, 30
COLOR 3
PLOT 10, 50
DRAWTO 159, 50
COLOR 0
PLOT 10, 70
DRAWTO 159, 70

Did you notice that nothing appeared to happen after you performed the last set of instructions? That is due to the fact that the fourth color, COLOR 0, is black. Now add this:

PLOT 80, 0
DRAWTO 80, 95

Now you can see COLOR 0 as it cuts through the other colors.

While working in the various graphics modes, you may encounter an ERROR 141. This simply means you have PLOTted or drawn to a point beyond the range of the mode in which you are working. In GRAPHICS 7, do not go beyond 159 in the column position, nor should you go beyond 95 in the row position. Otherwise you will end up with an ERROR 141.

CAUTION: row positions 80 through 95 are NOT displayed in GRAPHICS 7, but use of these positions does NOT generate an error.

Let's try to write a program using graphic mode 7. What we want to do is to create a bar graph that will indicate a person's physical, mental and emotional well-being. We will need to use GRAPHICS 7, COLOR, PRINT, INPUT, and other statements and commands discussed earlier in this book.

First, since we know we want to make a bar graph, we will need to get into graphics mode 7. Begin by pressing {SYSTEM RESET}.

Instruction:

You type:

NEW

100 GRAPHICS 7

Next, we want to formulate a PRINT statement that will indicate the intent of our program. You might choose something like this:

You type:

110 PRINT "ON A SCALE OF 1 TO 100, RATE YOUR"

120 PRINT "PHYSICAL, MENTAL, AND EMOTIONAL STATES"

130 PRINT "FOR TODAY."

SPECIAL NOTE: The reason we have chosen to write our PRINT statement on three separate lines is for a better format. All of our information would fit on three display lines, our legal limit for a program line. However, when the computer executed our program, some of the words would be divided. These divisions would not necessarily follow the syntax rules of English. To avoid user confusion, we divided our PRINT statement so that no words would be divided in a confusing manner.

Now we are going to put in a pair of statements that will allow some special features to be added to the program later.

Instruction:

You type:

200 LET DAYSTART = 10

210 LET DAY = 1

And then some statements which allow the program to interact with the user.

```
300 PRINT "PHYSICAL RATING, DAY "; DAY;
310 INPUT PHYSICAL
320 PRINT "EMOTIONAL RATING, DAY "; DAY;
330 INPUT EMOTIONAL
340 PRINT "MENTAL RATING, DAY ; DAY;
350 INPUT MENTAL
```

In statements 300 through 350, we have told the computer to PRINT messages and to wait for responses. Statements 400 through 480 will indicate COLOR, PLOTs, and DRAWTOs. These will allow us to form our bar graph.

Instruction:

You type:

```
400 COLOR 1
410 PLOT 0, DAYSTART
420 DRAWTO PHYSICAL, DAYSTART
```

These statements will draw an orange horizontal line (or "bar", hence "bar graph") on the screen. Since DAYSTART contains 10, the line will be drawn 10 units down from the top and will start at the left hand edge (PLOT 0, DAYSTART) and go right to the position specified by PHYSICAL (DRAWTO PHYSICAL, DAYSTART).

Now we will repeat similar statements for the EMOTIONAL and MENTAL portion of the graph.

Instruction:

You type:

```
430 COLOR 2
440 PLOT 0, DAYSTART + 2
```

This + 2 is to allow for separation between the lines of the graph.

Instruction:

You type:

```
450 DRAWTO EMOTIONAL, DAYSTART + 2
460 COLOR 3
470 PLOT 0, DAYSTART +4
480 DRAWTO MENTAL, DAYSTART +4
RUN
```

Answer the questions as they appear on the screen. Try this program several times varying your answers.

If you wanted to chart a person's physical, emotional and mental states for the entire week, what lines would you need to add? There are many attributes which make up a good programmer. One of those necessary qualities is insight. The above program is relatively straight forward. However, in order to expand the program, insight is necessary. You will recall that in the beginning of the program (lines 200 and 210), we indicated that these lines would allow features to be added. Now that we want to include the entire week on our bar graph, DAY and DAYSTART are very necessary.

Instruction:

You type:

```
500 LET DAYSTART = DAYSTART + 10
510 LET DAY = DAY + 1
520 IF DAY <= 7 THEN GOTO 300
```

In line 500 we have added a nonalgebraic expression which will increase the spaces on the graph. This is necessary so that the line of the graph will be distinct. Line 510 is another nonalgebraic equation which permits the number of days to increase by one each time line 510 is executed. Again, this permits a separation of groups of lines so that each day's rating will be distinguishable from the other days. Finally, the conditional statement in line 520 forces the repetition of lines 300 to 510 until 7 days have been completed. When DAY is equal to 7, the program will stop.

Exercises:

- 1) Write a program which will draw random lines on the screen, changing colors randomly.
- 2) Write a program which will draw random rectangles on the screen, changing colors randomly.
- 3) Write a program which will draw concentric boxes on the screen, changing colors randomly.

CHAPTER XV

EDITING FEATURES: THE SCREENING PROCESS

Glossary:

{DELETE} -----	This key will "erase" any letter you type by accident; it moves the cursor one space to the left each time you hit the key, removing the letter, symbol, or number it replaces.
{BREAK} -----	A key which when pressed stops whatever the computer was doing.
{CTRL} -----	This is the control key and when it is used in conjunction with various other keys it allows the user to edit program lines.
{←} -----	A key which when used in conjunction with the control key will move the cursor one character to the left without erasing the characters over which it passes.
{→} -----	A key which when used in conjunction with the control key will move the cursor one character to the right without erasing the characters over which it passes.
{INSERT} -----	A key which when used in conjunction with the control key will add a space to any part of a display line and at the same time will move the rest of the line one space to the right.
{ ↑ }	A key which when used in conjunction with the control key will move the cursor up one display line without erasing characters over which it passes.
{ ↓ }	A key which when used in conjunction with the control key will move the cursor down one display line without erasing the characters over which it passes.
{TAB}	Moves the cursor one tabular position to the right. The number of spaces between tabular positions is usually eight.
{SPACE BAR}	Moves the cursor one space to the right and replaces the character over which it passes with a blank space.

{CTRL} {1}	Allows the user to freeze a listing of a program.

{CTRL} {2}	Produces a bell sound.

{ESC}	Allows screen controls to take place when a program is running.

{CAPS/LOWR}	A key which allows the user to use lower as well as upper case letters.

```

-----
| Please note: {CTRL} and any |
| other key means to hold down |
| the {CTRL} key and at the same |
| time press the other key. For |
| example, in the glossary above |
| {CTRL} {1} means press {CTRL} |
| and at the same time press |
| {1} key. |
|-----|

```

Throughout this book we have discussed many of the special features of your Atari Home Computer. As you have learned to program BASIC, you have made mistakes. Mistakes are part of the learning process; and as your computer programming expertise continues to grow, your frustration at simple errors will also increase. This chapter is intended to explain all of the editing features of your Atari Home Computer and, in doing so, lessen those frustrations caused by simple typographical errors.

This chapter is divided into three sections. The first section discusses changes that can be made before the {RETURN} has been pressed. The second section explains the changes that can be made after the {RETURN} has been pressed. The final part of this chapter describes ancillary features.

Some keys which make it possible to correct errors have already been discussed. For the sake of completeness, we will review them here.

BEFORE {RETURN} IS PRESSED

The {BACK SPACE} key will "erase" any character you type. It moves the cursor one space to the left each time you hit the key. If you hold down this key or any of the other keys, the Atari Home Computer will automatically repeat the key function until you release the key. If you notice a mistake in a line, simply press the {BACK SPACE}. Back up to the mistake, and then correctly retype the line.

Instruction:

You type:

10 PINT "HELLO"

You know that the computer will not accept PINT. Press the {BACK SPACE} 11 times. Change PINT to PRINT and correctly retype the rest of the line. If that is too much work, you can simply press the {BREAK} key and the entire line will be eliminated.

Another way to change the line is by using the {CTRL} and {<--} keys. Located below the {CLEAR} key and the {DELETE BACK SPACE} key are four keys which have black arrows in white squares. These keys when pressed along with the {CONTROL} key will move the cursor one space in the direction that the arrow is pointing. As the cursor moves it does not erase the characters over which it passes.

Type the incorrect line again. Press {CTRL} and {<--} 11 times. The cursor should be on the letter I. Then press {CTRL} and {INSERT} once. When used in conjunction with the {CTRL} key, the {INSERT} key will add a space to any part of a display line and at the same time will move the rest of the line one space to the right. Type in R.

The changes you have made are on the display line only. If you want the changes also to be made in the computer's memory you must press the {RETURN} key. Be sure the line is typed correctly. Press {RETURN} and then type RUN.

Instruction:

You type:

10 PRINT "SAMM IS SXTY AND STLL SEXY"

To change this line, first press the {CTRL} and the {<--} key twenty-two times. Each time these keys are pressed, the cursor simply moves one space to the left and passes over, without erasing, each character.

The cursor should be on the second M. By pressing {CTRL} and the {DELETE} at the same time, all the characters on the right side of the cursor are moved one space to the left.

Using the {CTRL} and the {-->} keys, move the cursor 5 spaces to the right. The cursor is now on the X. To add one space before the X, press {CTRL} and {INSERT} once, and simply type in the missing I.

Press {CTRL} and {-->} 8 times, placing the cursor on the first L. Once again press {CTRL} and {INSERT} and add the missing I. Remember to press {RETURN} once the corrections are made to your satisfaction. Although the changes are present on the display screen, they are not recorded in the computer's memory until the {RETURN} key is pressed.

Sometimes a computer line will exceed a display line. Editing changes may be made before the {RETURN} key is pressed.

Instruction:

You type:

10 "SAM IS SIXTY AND STILL SEXY. HIS TNNIS SERVE TINKS.
SAMM DON'T CARE. HE STILL LUVES THE GAME."

In the above example, the words may break in inappropriate places. Please just ignore the English syntactical errors and concentrate on the editing features.

There is no correct order in which to make changes. If you decide to complete the changes in another way, that is fine. There are several different options. The following is but one of the possible methods. Begin by pressing {CTRL} and {<--} 15 times. The cursor should be on the U in the word LUVES. To change the U to an O, simply type O. The O will replace the U.

Using {CTRL} and {^} move the cursor up one display line. Like the other arrow keys, the {^} when used in conjunction with the {CTRL} key will move the cursor up one display line without erasing the characters over which it passes.

Next move the cursor one space to the right by using the {CTRL} and {-->}. The cursor should now be on the letter T in the word TINKS. Press {CTRL} and {INSERT} once and add the S before TINKS. Move the cursor 12 spaces to the left by using the {CTRL} and {<--}. The cursor should be on the first N in TENNIS. Then press {CTRL} and {INSERT} once and add the E.

Next, move the cursor 23 spaces to the right by using the {CTRL} and {-->} keys. The cursor should be on the second M in SAMM.

Another method you might try here is to use the {TAB} key. The {TAB} key moves the cursor one tabular position to the right. The number of spaces between tabular positions is usually eight. Press the {TAB} key 3 times. You may have to adjust the cursor from there.

The cursor should now be located on the N in DON'T. Press {CTRL} and {INSERT} twice and type in E and S.

Have all the corrections been made? If so, press {RETURN}. Notice where the cursor is now located. It is at the beginning of the next computer line; not the next display line. Remember, pressing {RETURN} stores our PRINT statement into the computer's memory. Even though we have used almost three full display lines, we have used only one computer line. The cursor indicates that the computer is waiting for our next program line or command. Type RUN. The corrections have been recorded.

Notice that in our PRINT statement all the words are syntactically correct, but not when we RUN the program. Now let's make the changes so our program will RUN properly.

AFTER YOU HAVE PRESSED RETURN

When making changes after the {RETURN} key has been pressed, it is necessary to go back to the program line -- not the area of the monitor where the computer displayed our program.

Press the {CTRL} and {^} 8 times. Press the {CTRL} and {-->} 3 times. The cursor should be on the blank space between HIS and TENNIS. Press {CTRL} and {INSERT} 4 times. This will add 4 spaces between HIS and TENNIS. This should take care of the irregular break in the word TENNIS.

Now using the {CTRL} and {v}, move the cursor down one line. The {CTRL} and {v} function in the same manner as the other arrow keys. The cursor moves down one display line without erasing any characters each time {CTRL} and {v} are pressed simultaneously. The cursor should be located on the blank space before HE.

Press {CTRL} and {-->} 7 times, and press {CTRL} and {INSERT} once. Now press {RETURN} again. Remember for changes to take place in the computer's memory as well as on the display screen, you must press {RETURN}. Since there is so much information on the screen, press {SHIFT} and {CLEAR}. This will remove all material from the display screen without erasing the computer's memory.

Instruction:

You type:

LIST
RUN

Notice anything strange? Yes. There is a period at the beginning of the line. Not exactly where it should be. Again, go back to the PRINT statement. Change the line by adding four spaces before the word CARE. You may do this using the {CTRL} key and the various arrow keys to move the cursor. Then as we have done in previous examples, you may use the {CTRL} and {INSERT} keys to add the necessary spaces.

Again press {SHIFT} and {CLEAR} to remove all the material that is currently on the screen. Type:

LIST
RUN

What happened this time when you ran the program? Look at the word "GAME". Why is there only a G and not the whole word GAME? While we were adding spaces to correct the syntactical errors, we added more spaces to our computer line. Blank spaces are counted by the computer in the same manner that it counts any character -- letter, number, punctuation mark or special symbol. When we added extra spaces we exceeded our legal line limit of 114 characters. The computer automatically cuts off any characters after the legal limit.

ADDITIONAL FEATURES

As you probably have noticed, the {CONTROL} key used in conjunction with other keys gives the Atari Home Computer many additional editing features. Some have already been mentioned, and although several others do exist, we will only discuss two more {CONTROL} features here. If you have written a very long program and want to check various parts of the listing, you can stop the listing by pressing the {BREAK} key. Although effective, this method is somewhat hit and miss. By pressing {CTRL} key and {1} you can freeze the listing of a program. When you type {CTRL} and {1} again, the listing will continue. {CTRL} and {1} allows you to review a program listing at your own pace.

Another {CTRL} feature is used in conjunction with {2}. This produces a bell sound. Used within a PRINT statement, {CTRL} and {2} can act as an alarm to let you know when the computer is PRINTing something in particular. It is also an easier way to produce a sound than going through the sound registers.

In order to use the {CTRL} {2} within a program, you must also use the {ESC} key. {ESC} stands for escape. This term dates back to the time when teletypes were commonly used as computer terminals. Although that is not the case today, the name has stuck just like the term PRINT has stuck. It is located in the top row to the left of the {1} key. The escape key, unlike the {CTRL} and {SHIFT} keys, is pressed and released before another key is pressed.

Essentially, the {ESC} key works in a similar manner to program line numbers. The line numbers defer the operation of statements until the program is RUN. The {ESC} key defers the operation of the editing keys until a program is RUN. There is one exception: {ESC} {CTRL} {1} cannot be edited into a program.

Instruction:

You type:

```
10 LET I = 1
20 IF LET = I + 1
30 IF I < 100 THEN GOTO 20
40 PRINT "[ESC] [CTRL] {2} I'M FINISHED COUNTING"
```

Remember! {CTRL} {2} means hold down {CTRL} while typing {2}.

RUN

Be sure to watch the screen, and listen. When did the bell ring?

Try another program using the {ESC} key.

Instruction:

You type:

```
10 PRINT "{ESC} {CLEAR}"
20 PRINT "AHA! THE SCREEN IS CLEARED!"
30 PRINT "GREAT {ESC} {<--} {ESC} {<--} {ESC} {<--}
   {ESC} {<--} {ESC} {<--} {ESC} {DOWN ARROW}"
```

In the beginning of this book, we told you that your Atari Home Computer normally displays UPPER-CASE letters. The computer will accept instructions written in lower-case letters; and there may be times when, for appearance sake or for clarity, you would like to use lower-case letters.

Find the {CAPS/LOWR} key located directly below the {RETURN} key. If you press this key once, you will be able to type lower-case letters, numbers, some punctuation marks and the arithmetic operation symbols. If you now press either {SHIFT} key and at the same time another key, you will be able to produce upper-case letters (capital letters) of any of the characters shown on the upper half of the keytop on the keyboard. If you want to return to upper-case letters, press the {SHIFT} and {CAPS/LOWR} keys at the same time.

A special feature of your Atari Home Computer keyboard is a special set of graphic characters that appear only when the {CTRL} key is pressed at the same time as another key. Using these graphic keys, you can create interesting picture graphics or designs.

This mode will work either with your BASIC cartridge or without it. If you want to stay in this graphic drawing mode, press the {CTRL} key and the {CAPS/LOWR} key at the same time. This procedure will lock the keyboard into the "graphic drawing mode". In order to return the keyboard back to normal, press one of the {SHIFT} keys at the same time as you press the {CAPS/LOWR} key.

You may have noticed on your computer keyboard one key which has the Atari logo on it. This key switches the video mode from the normal display--blue background with white lettering--to inverse video. In inverse video the characters are blue with a white background.

As you become familiar with these various editing features, you will find them very useful. As with other facets of the computer, you will probably make some mistakes, erasing or deleting lines which you wanted to save. Like anything else, practice will increase your skill and lower your frustration level.

CHAPTER XVI

IF REVISITED: THEN WE CAN DO ANYTHING

Glossary

IF...THEN -----	A conditional statement which causes the computer to make a decision.
END ---	A statement which tells the computer to suspend execution of the program.
: ---	A punctuation mark used in BASIC XL which allows the programmer to use multiple statements on a single program lines.

When we discussed IF...THEN in Chapter VII we may have given you the impression that an IF...THEN statement was always followed by a GOTO statement or a PRINT statement. Not so. Any statement may follow an IF...THEN statement.

Examples:

```
10 IF A = 100 THEN PRINT "TERRIFIC"
10 IF TREES = OAK THEN GOTO 100
10 IF BONUS = 1500 THEN LET SALARY = 25000
10 IF A > B THEN INPUT D
```

In each of the program lines above, the IF part begins a conditional statement. The computer must decide whether the statement is true or not. Assuming the statement is true, the computer does whatever is specified in the rest of the program line.

It may PRINT or GOTO. It may also initialize a variable or change the value of a variable. Or, the computer may wait for data to be entered from the keyboard. BASIC XL also permits the programmer to use an IF...THEN statement followed by just a line number.

Instruction:

You type:

```
10 PRINT "WHAT WAS YOUR MOST RECENT GRADE POINT AVERAGE"
20 INPUT AVERAGE
30 IF AVERAGE < 3 THEN 60
40 PRINT "I'M PROUD OF YOU"
50 END
60 PRINT "YOU BETTER STUDY"
```

In the above program our first PRINT statement indicates the data desired and our INPUT statement allows the user to enter that data into the computer. We use an IF...THEN statement in line 30 in conjunction with only a line number. It is as though the GOTO is implied. The computer executes THEN GOTO and THEN line number in the same manner.

For the first time in the above program we have used an END statement. END tells the computer to discontinue the execution of a program and to return to direct mode. Often an END statement is the last statement in a program. BASIC XL does not require an END statement. However, an END statement is necessary to our above program.

What would happen without the END statement in line 50? If the user's grade point average was greater than 3, the computer would PRINT lines 40 and 60 which you may or may not desire. With our END statement, a user with a grade point average equal or greater than 3 would receive the message "I'M PROUD OF YOU". While the user with a grade point average of less than 3 would receive the message "YOU BETTER STUDY". The END statement will be used again in our next chapter. Just remember END discontinues the execution of a program and places the computer in direct mode.

Sometimes you may want to join two or more statements into one program line. By using a colon, you as a programmer may connect multiple statements on the same program line.

Examples:

```
40 IF X <> A THEN PRINT "TRY AGAIN": GOTO 10
10 IF G <= R AND B <> K THEN LET X=50: PRINT
   "THAT IS INCORRECT"
30 PRINT: PRINT: PRINT "LOTS OF PRINTS ON ONE LINE":PRINT:
   PRINT
```

Remember, it is perfectly legal and syntactically correct to add a colon in order to put more than one instruction on a program line. However, in joining statements you may not exceed the single program line limit of 114 characters. Using a colon may enable you to better organize your programs and to save time and space.

Throughout this book whenever we have named a variable, we have used a LET statement. We did this to remind you that we were dealing with a variable and not a mathematical expression. Although it is good form and

acts as a reminder, the LET is usually not necessary. In BASIC XL the LET is implied whenever a variable is named.

Of course, there is an exception. Should you want to use a statement word as the name of your variable, you must use the LET. For example:

```
10 LET LET= 15
20 LET PRINT= 235
30 PRINT PRINT, LET
```

Also, if your variable name BEGINS with a statement word, you MUST use LET.

```
30 LET COLOR0 = 3
40 COLOR = COLOR0
50 LET LETTER = 26
```

Exercises:

1) Reggie Smith's salary is based on sales. He is paid \$1000 unless his sales are over \$20,000, in which case he is paid \$2000. Write a program which will PRINT his salary.

2) Modify the above program to include deductions for federal taxes and Social Security. Use a colon to join computer statements on one line. (Use arbitrary values for the taxes, if you like.)

Chapter XVII

SUBROUTINES: CALLING FOR HELP

Glossary:

Subroutine A statement or a group of statements

 within a computer program, yet
 distinguishable from the rest of the
 program, which performs a separate and
 complete function.

GOSUB A statement that tells the computer to

 execute a subroutine. This statement is
 always followed by a line number and is
 always paired with a RETURN command.

RETURN A statement which ends a subroutine and

 tells the computer to go back to the next
 command in the main body of the program.

As your ability to program increases, the length of your programs will also increase. Long programs can become cumbersome. They are difficult to read and their logic may be too hard for a beginner to decipher. In analyzing a long program, it is helpful to break it into functional parts. You may find some repetition in the program. Perhaps a group of commands have been repeated in several places.

Instruction:

You type:

```
10 LET SCORE = 0
20 PRINT "ONE CUP EQUALS 1) 8 OZS. 2) 16 OZS. 3) 32 OZS."
30 LET CORRECTANSR = 1
40 INPUT ANSWER
50 IF ANSWER <> CORRECTANSR THEN PRINT "THE CORRECT
   ANSWER IS"; CORRECTANSR
60 IF ANSWER = CORRECTANSR THEN PRINT "THAT'S RIGHT":
   LET SCORE = SCORE + 1
70 PRINT "ONE QUART EQUALS 1) 18 OZS. 2) 24 OZS. 3) 32 OZS"
80 LET CORRECTANSR = 3
90 INPUT ANSWER
100 IF ANSWER <> CORRECTANSR THEN PRINT "THE CORRECT
    ANSWER IS"; CORRECTANSR
110 IF ANSWER = CORRECTANSR THEN PRINT "THAT'S RIGHT": LET
    SCORE = SCORE + 1
120 PRINT "ONE GALLON EQUALS 1) 64 OZS. 2) 128 OZS. 3) 200
    OZS"
130 LET CORRECTANSR = 2
```



```

140 INPUT ANSWER
150 IF ANSWER <> CORRECTANSR THEN PRINT "THE CORRECT ANSWER
    IS"; CORRECTANSR
160 IF ANSWER = CORRECTANSR THEN PRINT "THAT'S RIGHT": LET
    SCORE = SCORE + 1
290 PRINT "THE TEST IS OVER.  YOUR SCORE IS"; SCORE

```

```

-----
| By now, it would be nice if you |
| could read a program such as the |
| above and understand most of what |
| it does.  If you can't do this |
| yet, do not worry about it.  Type |
| the program in and RUN it.  But |
| keep trying to "read" programs, |
| since being able to do so will |
| make it easier to work with both |
| this book and computer magazines! |
|                                     |
-----

```

In the program above, first notice how many program lines we used. Did you find any repetitions? Yes, lines 40, 50, and 60 are the same as lines 90, 100 and 110 and the same as lines 140, 150, and 160. Wherever you find the repetition of program lines, you have the basis for a subroutine.

A subroutine consists of a group of lines which usually perform a particular function and are terminated by a RETURN statement. Most often, a subroutine is used to execute a specified task by allowing the task to be referenced from more than one location in the main program.

A subroutine is a statement or group of statements within a computer program which performs a separate and complete function. Subroutines are distinguishable from the rest of the program. Usually, but not always, a subroutine is placed at the very end of the program.

A GOSUB statement is closely related to a GOTO statement. Both are followed by line numbers. The one major difference is that a GOSUB makes the computer remember where it "left off" before it goes to its target line, which should be a subroutine. Also, a RETURN is always the last executed statement of a subroutine. The RETURN statement causes the computer to go back to the main body of the program and execute the statement following the GOSUB statement.

Although a program may contain numerous GOSUB statements (including, usually, several GOSUBS to the same subroutine), the computer always remembers at which line

it left the main program. When the computer encounters the RETURN, it knows exactly where it left off. It goes back to its place and continues to execute the next statement following the GOSUB.

Let's return to our example. We used 17 computer lines to give our quiz and to keep the user's score. The discussion which follows is organized by line numbers to make it easy for you to refer to the program.

line 10 - The score a person receives when taking a test is important. Thus, we will have the computer keep track of the score for us. In this line, we will initialize our variable score at 0. We do this because at the beginning of a test, there is no grade. Once a question has been answered correctly or incorrectly there is a positive or negative score.

```
-----  
| NOTE: In BASIC, it is actually |  
| often not necessary to initialize |  
| a variable to 0. The RUN command |  
| automatically initial-izes all |  
| variables to 0. All that is nec- |  
| essary is to name the variable. |  
-----
```

line 20 - PRINT statement indicates our first question.

line 30 - the value of our variable CORRECTANSR is set at 1. We did this because the answer to the question is 1.

line 40 - the INPUT statement allows the user to enter data.

line 50 - forces the computer into making a decision. If the answer is incorrect, PRINT the correct answer.

line 60 - forces the computer into making a decision. If the answer is correct, PRINT a remark which praises the user and increases the user's score by 1.

line 70 - PRINT statement indicates our second question.

line 80 - the value of our variable CORRECTANSR is set at 3. Again, we did this because the answer to the question is 3.

- line 90 - the INPUT statement allows the user to enter data.
- line 100 - forces the computer into making a decision. If the answer is incorrect; PRINT the correct answer.
- line 110 - forces the computer into making a decision. If the answer is correct, PRINT a remark which praises the user and increases the user's score by 1.
- line 120 - PRINT statement indicates our third question.
- line 130 - the value of our variable CORRECTANSR is set at 2. We did this because the answer to the third question is 2.
- line 140 - the INPUT statement allows the user to enter data.
- line 150 - forces the computer into another decision. If the answer is incorrect, PRINT the correct answer.
- line 160 - forces the computer to determine if the answer is correct. If so, PRINT a remark which praises the user and increases the user's score by 1.
- line 290 - PRINT statement indicates the end of our program and displays the user's score.

Notice that lines 20 and 30 resemble lines 70 and 80 and lines 120 and 130. Although these statements are similar, they are not exactly the same.

Only lines which are exactly the same can be used for a subroutine.

But look at lines 90, 100, 110. Aren't they exactly the same as lines 40, 50, 60, and 140, 150, 160? Yes, they are. These could be the basis for our subroutine. The last line of our program is self explanatory.

Let's rewrite our quiz program using a subroutine. Lines 10, 20, 30 need not be re-entered. They are correct as is.

Instruction:

You type:

40 GOSUB 400
50 {RETURN}
60 {RETURN}
90 GOSUB 400
100 {RETURN}
110 {RETURN}
140 GOSUB 400
150 {RETURN}
160 {RETURN}
300 END

400 INPUT ANSWER
410 IF ANSWER <> CORRECTANSR THEN PRINT "THE CORRECT
ANSWER IS" ; CORRECTANSR
420 IF ANSWER = CORRECTANSR THEN PRINT "THAT'S RIGHT"
LET SCORE = SCORE + 1
430 RETURN

LIST

The listed program should look like this:

10 LET SCORE = 0
20 PRINT "ONE CUP EQUALS 1) 8 OZS. 2) 16 OZS. 3) 32
OZS."
30 LET CORRECTANSR = 1
40 GOSUB 400
70 PRINT "ONE QUART EQUALS 1) 18 OZS. 2) 24 OZS. 3) 32
OZS."
80 LET CORRECTANSR = 3
90 GOSUB 400
120 PRINT "ONE GALLON EQUALS 1) 64 OZS. 2) 128 OZS. 3)
200 OZS."
130 LET CORRECTANSR = 2
140 GOSUB 400
290 PRINT "THE TEST IS OVER. YOUR SCORE IS" ; SCORE
300 END
400 INPUT ANSWER
410 IF ANSWER <> CORRECTANSR THEN PRINT "THE CORRECT
ANSWER IS" ; CORRECTANSR
420 IF ANSWER = CORRECTANSR THEN PRINT "THAT'S RIGHT"
LET SCORE = SCORE + 1
430 RETURN

Again, let's analyze each line; but this time let's do
so in the order in which BASIC actually executes the
program:

line 10 - sets the value of the variable score at 0.

line 20 - the PRINT statement indicates our first question.
 line 30 - sets the value of the variable CORRECTANSR at 1.
 line 40 - sends the computer to our subroutine.
 line 400 - allows the user to enter data.
 line 410 - forces the computer into making a decision. If the answer is incorrect, PRINT the correct answer.
 line 420 - forces the computer into making a decision. If the answer is correct, PRINT a remark which praises the user and increases the user's score by 1.
 line 430 - causes the computer to go back to the main body of the program and to execute the next statement after the GOSUB 400. The first time through the subroutine, the computer will go back to line 40.
 line 70 - the PRINT statement indicates our second question.
 line 80 - sets the value of the variable CORRECTANSR at 3.
 line 90 - sends the computer to our subroutine. Lines 400 through 430 are repeated, exactly as after line 40.
 line 120 - the PRINT statement indicates our third question.
 line 130 - sets the value of the variable CORRECTANSR at 2.
 line 140 - sends the computer to our subroutine. Lines 400 through 430 are repeated, exactly as after line 40.
 line 290 - the PRINT statement that indicates the test is over and the user's score.
 line 300 - tells the computer not to go any further.

The numbers may be a bit confusing, but notice how much easier it is to read and to understand the organization of our new program. In addition to the subroutine, we also added an END statement. This was necessary so that

when the computer reached the last program line, it did not try to execute our subroutine. Remember without the END statement, the computer would execute each line in numerical order. Eventually, the computer would reach line 400 and would wait for the user to INPUT an answer. Can you see all the problems this would cause? Even if the user types in an answer, what is the correct answer? The computer would eventually PRINT ERROR.

A diagram of how the program would RUN might look like this:

```
10 20 30 40      70 80 90      120 130 140      300
400 410 420 430   400 410 420 430   400 410 420 430
```

Subroutines may be as long or as short as you desire. They provide an efficient means of organizing a program that repeats a specified task, and in general, subroutines make programs easier to understand.

Exercise:

1) Add two more questions to the above program.

Chapter XVIII

BETTER LOOPS: WHAT'S THE NEXT STEP

Glossary:

FOR ---	A statement which indicates a sequence that the computer is to complete a number of times. FOR is always used in conjunction with NEXT.
NEXT ----	A statement which when used with FOR forms a loop. This loop provides an automatic method for making the computer count.
STEP ----	Is the same as an increment. STEP is used with a FOR statement when you want the computer to count by an increment other than positive one.
Negative Number -----	A number whose value is less than zero. A negative number is associated with a minus sign.

In reading this book, you have probably noticed that there is often more than one method for obtaining a desired output. At first, we used two variables as a counter. Later, we introduced a nonalgebraic equation ($LET\ X = X+1$) as a counter. In previous examples we used LET, IF...THEN and GOTO statements to produce a counter within our program. Two additional statements which are often used together as a counter are FOR...NEXT.

The statements FOR...NEXT form a loop. FOR...NEXT must always be used in conjunction with each other. If you use the NEXT without the FOR, the computer will produce ERROR 13-NEXT WITH NO MATCHING FOR. If you use FOR without a NEXT, the computer will be unable to find the end of the loop. The FOR...NEXT loop acts as an automatic counter. The FOR statement is followed by a numeric variable.

Instruction:

You type:

```
10 FOR Y = 1 TO 10
20 PRINT Y, "WATCH ME COUNT"
30 NEXT Y
```

In line 10 above, the FOR is followed by our numeric variable Y. The one (1) indicates the lower limit of the variable, or the value of Y when the loop begins. In other words, the numeric value following the equal sign initializes the variable.

The number following TO indicates the upper limit of the value of the variable. When the variable reaches its upper limit, in this case 10, the loop is completed.

FOR	Y	=	1	TO	10
	NUMERIC		INITIALIZES		THE UPPER LIMIT;
	VARIABLE		THE VARIABLE		THE VALUE OF Y
					WHEN THE LOOP IS
					COMPLETED.

Between the FOR and the NEXT is a statement. Actually, any number of lines and statements could be placed between the FOR and the NEXT. These statements indicate what the computer is to do. In our example, we want the computer to PRINT the value of Y and the string "WATCH ME COUNT". The number of times that the statement or statements is/are executed depends on the number of times the FOR statement indicates that the loop is to be completed. The variable in the NEXT statement must be the same as the variable in the FOR statement. The NEXT statement is always the last statement in the loop.

Instruction:

You type:

Computer Responds

```
10 FOR A = 1 TO 3
20 PRINT A, A+A, A*A
30 NEXT A
RUN
```

1	2	1
2	4	4
3	6	9

In the above program, we have initialized our variable A and set its upper limit at 3. Our PRINT statement indicates what we want to see on the screen is the value of A, the sum A+A, and finally the product A*A. Our NEXT statement completes the loop and sends the computer back to line 10 until it completes the loop the prescribed number of times.

In the above examples each time the loop was completed, the variable increased its value by one. It repeats the process until the maximum value allowed by the FOR statement.

What do you do if you want to increase the value of the FOR variable by more than one? What do you do if you want to decrease the value of the FOR variable?

The way to make the computer count by increments other than one is to use the keyword STEP. STEP, followed by a number, tells the computer to count in a specific increment.

Instruction:

You type:

Computer Responds

10 FOR M = 3 TO 15 STEP 3
20 PRINT M
30 NEXT M
RUN

3
6
9
12
15

The STEP will force the computer to count by 3's. If you are counting by anything other than positive one, STEP becomes part of a FOR statement. (In fact, STEP may only be used in a FOR statement.)

Examples:

10 FOR X = 10 TO 100 STEP 5 (counts from 10 to 100 by five's)

10 FOR S = 2 to 40 STEP 2 (counts to 40 by two's)

COMMENTARY:

In our discussions to this point, we have used only positive numbers. We have counted from zero or one in a positive manner, ascending the number line.

However, for every positive number there is also a negative number. Look at the number line below.

<-----0----->
-5 -4 -3 -2 -1 . 1 2 3 4 5

A negative number is named by a numeral and a minus (or "negative") sign. Whenever the direction of movement on a number line is to the right of the point of origin, we say the movement is in a positive or ascending direction. Whenever the direction of movement on a

number line is to the left of the point of origin, we say the movement is in a negative or descending direction. Remember, the point of origin on the number scale and the direction of the movement determines positive and negative values.

Example:

<-----+----->

11 12 13 14 15 16 17

In the above example 15 is in a positive direction from our point of origin; 13 is in a negative direction.

We often discuss negative numbers in terms of cities and their altitudes. Sea level is our point of origin. The altitude of a particular city is determined by its relationship to sea level. The city has a positive altitude if it is above sea level. Denver, Colorado, is often called the mile high city because it is approximately 5280 feet above sea level. On the other hand, Death Valley, California, has a negative value. It is approximately 280 feet below sea level.

When using a FOR...NEXT loop, it is possible to move in a negative direction as well as a positive one. Again, in order to count in an increment other than positive one, STEP is added to the FOR statement. By using a negative number, you indicate that the counting is to take place in a negative manner.

Examples:

10 FOR C=10 TO 1 STEP -1 (counts backwards from 10 to 1)

10 FOR D = 100 TO 0 STEP -2 (counts backwards from 100 by two's)

Notice in our FOR statements that the upper limit of our variable is stated first and then the lower limit. When using STEP to decrease the value of the FOR variable, the larger value must be stated first, then the smaller value.

Exercises:

- 1) Write a program to PRINT your name 10 times.
- 2) Write a program which will count backwards from 10. Instead of zero have the computer print "BLAST OFF".

3) Write a program which will count from 0 to 100 by fives.

4) Write a program which will PRINT all numbers and their squares, starting at 1 and continuing until it reaches a maximum number, which should be specified by a user INPUT.

Chapter XIX

FOR LOOPS REVISITED: ANOTHER STEP UP

Glossary:

FOR A statement which indicates a sequence that
--- the computer is to complete a number of
 times. FOR is always used in conjunction
 with NEXT.

NEXT A statement which when used with FOR forms
---- a loop. This loop provides an automatic
 method for making the computer count.

Nest To fit together or within one another, as
---- boxes, mixing bowls or small tables (verb).

Nest An assemblage of things lying or set close
---- together or within one another as a nest of
 tables (noun).

A FOR ... NEXT loop is a useful programming tool.
However, it can become more powerful when used in
conjunction with other FOR ... NEXT loops.

The process of placing FOR ... NEXT loops within one
another is called "nesting". Nesting is a desirable
and beautifully structured programming capability.
Remember, though, that the innermost level of a nest
must be completed before the next level outward can be
executed.

To illustrate the concept of nesting, we will start with
a non-computer example. Imagine that it is your job to
inspect strawberries for quality. There are twenty
strawberries in each basket and 8 baskets in each of
four crates. You must first open a crate, then remove a
basket, and finally, remove each strawberry and examine
it. After you have inspected each strawberry, you would
have to replace them in the basket. After you have
inspected each strawberry in each basket, you would
replace the baskets into the crate. Then, you would
follow the same procedure with the next crate.

If we were to write the above example in outline form,
it would look like this:

- I. For each of four crates and
 - A. For each of 8 baskets in each crate and
 1. For each of 20 strawberries in each basket,
 - i. You must inspect a single strawberry and
 - ii. Put it back in its basket. Then,
 2. When all 20 strawberries are checked and replaced, replace the basket in the crate. Then,
 - B. When all 8 baskets are checked and replaced, put the finished crate aside. Then,
- II. When all four crates are checked, you are done.

Now we will "diagram" the same procedure in a paraphrase of BASIC.

```

FOR CRATE = 1 TO 4
  FOR BASKET = 1 TO 24
    FOR STRAWBERRY = 1 TO 20
      INSPECT A SINGLE STRAWBERRY
      REPLACE IT IN ITS BASKET
    NEXT STRAWBERRY
  REPLACE THE BASKET IN THE CRATE
  NEXT BASKET
SET THE CRATE ASIDE
NEXT CRATE

```

Notice in our example above, our FOR and NEXT loops are paired from the inside out. First, our STRAWBERRY loop is completed, then the BASKET, and finally the CRATE.

Was this example a legitimate BASIC program? No, because BASIC doesn't understand "REPLACE...", "INSPECT...", or "SET THE...". The FOR...NEXT statements, though, were completely legitimate.

Suppose we modify the task a little. Instead of inspecting strawberries, let's show a star on the screen for each strawberry. And let's PRINT the words "BASKET" and "CRATE" as we complete each task. Our program might look like this:

Again, this is perfectly legal as long as you nest FOR...NEXT loops from the inside out.

Instruction:

You type:

NEW

```
10 FOR CRATE=1 TO 4
20 FOR BASKET = 1 TO 8
30 FOR STRAWBERRY =1 TO 20
40 PRINT "*";
50 NEXT STRAWBERRY
60 PRINT "BASKET"
70 NEXT BASKET
80 PRINT: PRINT "CRATE": PRINT
90 NEXT CRATE
RUN
```

What does this program produce? How many times does the computer execute the FOR CRATE and NEXT CRATE loop? Which loop is executed more often, the FOR CRATE and NEXT CRATE or the FOR BASKET and NEXT BASKET or FOR STRAWBERRY AND NEXT STRAWBERRY loop?

Let's analyze this program:

line 10 - FOR is followed by our numeric variable CRATE, and the range of CRATE is set. The starting value of CRATE will be 1. It's final value will be 4.

line 20 - We begin another FOR...NEXT loop. BASKET is our second numeric variable. It's range is 1 to 24.
line 30 - Is the same as line 20 except STRAWBERRY is the name of the numeric variable and its value is 1 to 20.

line 40 - Is a PRINT statement. Each time the computer executes line 40, it will produce a *. Notice the semicolon. Remember the semicolon suppresses the start of a new line.

line 50 - Properly completes our FOR STRAWBERRY loop with NEXT STRAWBERRY. FOR...NEXT loops are automatic counters. The NEXT STRAWBERRY command forces the computer to go back to line 30. The return to line 30 continues until the value of STRAWBERRY is >20.

line 60 - Tells the computer to PRINT the word BASKET and move the cursor to the next line. Because of the semicolon in line 40, the word BASKET is PRINTED next to twenty stars.

line 70 - Properly completes our FOR BASKET loop with NEXT BASKET. This time the computer is forced to return

to line 20. Notice that this forces the computer to re-execute our FOR STRAWBERRY and NEXT STRAWBERRY loop each time the value of BASKET changes. When BASKET is greater than 8, the computer will drop down to line 80.

line 80 - Tells the computer to PRINT a blank line. By using the colon, we are able to put more than one command on this line. As you will recall, a PRINT statement by itself will force the computer to PRINT a blank line. After the computer PRINTs a blank line, it will PRINT the word CRATE. And since we used another colon and another PRINT statement, the computer PRINTs an additional blank line.

line 90 - Properly completes our FOR CRATE loop with NEXT CRATE. The NEXT CRATE command forces the computer to go back to line 10. The return to line 10 continues until the value of CRATE is >4. When CRATE is > 4, the program will end.

Instruction:

You type:

```
NEW
100 FOR ROW = 1 TO 10
110 FOR STAR = 1 TO ROW
120 PRINT "**";
130 NEXT STAR
140 PRINT
150 NEXT ROW
RUN
```

The above program is essentially a variation of the previous program. Once again, in line 100 we begin our FOR ROW ... NEXT ROW loop. The value of the variable ROW will have a range of 1 to 10. That is, the initial value will be 1, and it's final value will be 10.

In line 120, we name our other numeric variable STAR. Notice that the initial value of STAR is 1 and subsequent values are determined by the value of the variable ROW.

Lines 130 to 150 are a repetition of lines 30 to 60 in our previous program.

If you were to play computer, the following would be the sequence of steps you would follow:

(line number)

1. (100) initialize the variable ROW at 1
2. (110) initialize the variable STAR at 1
3. (120) PRINT * without moving to the next line
4. (130) add 1 to STAR and go to line 110
5. (110) determine that STAR > ROW ; go to line 140 (the statement following NEXT STAR)
6. (140) PRINT a blank line adjacent to the * (just moves to the beginning of the next line)
7. (150) add 1 to ROW and go to line 100
8. (100) the value of ROW is less than 10 so continue with the next statement (line 110)
9. (110) set the value of STAR to 1
10. (120) PRINT * without moving to the next line
11. (130) add 1 to STAR and go to line 110
12. (110) the value of STAR (now 2) is still <= ROW (currently 2) so continue with line 120
13. (120) PRINT * on the same line as previous *
14. (130) add 1 to STAR and go to line 110
15. (110) the value of STAR (now 3) is > ROW so go to line 140
16. (140) PRINT a blank line next to the second * (just moves to the start of the next line)
17. (150) add 1 to ROW and go to line 100

.
.
.

This program will finish executing when the value of ROW is equal to 10. Each time the computer executes the outer loop (FOR ROW...NEXT ROW) once, it executes the inner loop (FOR STAR...NEXT STAR) 'ROW' number of times.

Exercise:

- 1) Write a program using several FOR and NEXT loops to create a multiplication table.

Chapter XX

STRING VARIABLES PART I: REMEMBERING WORDS

Glossary:

String Variable -----	Is a variable that is capable of containing a value consisting of words, letters, or characters (in contrast to a numeric variable, Chapter III, which can hold only numbers).
DIM ---	Is the statement used to tell the computer how much memory might be needed to hold the string value of a variable.
\$ ---	Is the symbol which must be used as the last character in the name of a string variable. The \$ symbols tells the computer that a particular variable is a string, not a numeric.
= ---	A comparison operator that can be used with strings.
<> ---	A comparison operator that can be used with strings.

As you have seen, computer programming with just numeric constants (such as: 3,7, etc.) was rather limiting. When we added numeric variables to our repertoire of programming statements, we were able to produce more useful programs.

Just as there were numeric constants, BASIC also uses string constants. Our use of strings up to this point has been rather limited. You will recall that a string is any group of characters inside quotation marks. In previous chapters we have used string constants only in our PRINT statements. Formerly used examples include:

```
PRINT "HELLO"  
PRINT "HI, MOM"  
PRINT "YOUR NAME"
```

It is also possible in BASIC to use string variables. When using a string variable, the variable takes the place of a string of characters, just as a numeric variable is used in place of a number. String variables' names are distinguished from numeric names by

always being followed by a dollar sign {\$}. The dollar sign {\$} indicates to the computer that the variable is NOT a numeric variable. To the computer X and X\$ are very different. When the computer encounters X, it expects X to have a numerical value. However, when the computer reads X\$, it expects the value of X\$ to be a string of characters.

Since strings come in any size (while numbers are all the same size as far as the computer is concerned), we may tell the computer the largest possible number of characters a given string variable might be allowed to contain.

Thus string variables differ from numerics is in the optional use of a DIMension statement. Before we can use a string variable to hold a string value, we must first indicate to the computer how long we want it to allow the string value to become. If we don't tell the computer how long a string value a particular variable must hold, BASIC XL assumes that it will hold 40 characters (enough for any string that can fit on a single screen line).

Just as a numeric variable can remember a single numeric value, so can a string variable remember a single string. (Remember the definition of a string is zero or more characters. Generally, when we see a string in a program it is enclosed in quotation marks, as in chapter I).

Instruction:

You type:

10 DIM B\$(5)

In the above DIMension statement, we told the computer that a string variable called B\$ might have as many as five characters.

You type:

20 LET B\$ = "JULIE"

This LET statement tells the computer that the value of the string variable B\$ is JULIE. Note that the quote marks are NOT part of the string. Just as in English, they are simply punctuations which show where the string begins and ends.

One note of caution: When you DIMension your string variable, be sure to allow enough room for all possible, reasonable answers. If, in the above example, you wanted B\$="WILLIAM", the computer would accept only the first five characters, and B\$ would contain "WILLI" again, without the quote marks. On the other hand, B\$ could = "SUE" or "JOE" or "MIKE" because they are less than the maximum number of characters specified in our DIMension statement.

In Chapter III we defined a numeric variable. Here is a summary of the differences between a numeric variable and a string variable. First, a numeric variable represents a number; a string variable represents a string--that is a group of characters or words. Second, you must DIMension a string variable, telling the computer how much memory to reserve for the string variable. Third, when using a string variable, you must distinguish the variable by following the letter or last letter in the variable name with a dollar sign (\$). Finally, just like other strings constants used previously in this book, you must enclose the value to be assigned to a string variable (i.e., in a LET statement).

All of this is particularly important in programs which use INPUT statements. As stated in Chapter VIII nothing will happen, after the execution of an INPUT statement, until the computer receives an acceptable answer. If you try to INPUT a string value where the computer is expecting a numeric, the computer will produce a message--usually an ERROR 8 - INPUT/READ. The INPUT statement did not receive the type of data it expected.

The reverse is NOT true. If you use numbers where the computer is expecting INPUT of a string value, the computer will accept the numbers, since a numeric character is no less a character than is a letter. However, no arithmetic operations can take place with those numbers. The computer treats them as a string.

Instruction:

You type:

```
10 PRINT "WHAT IS YOUR NAME";
20 INPUT N$
30 PRINT "HELLO," ; N$
40 PRINT "HOW OLD ARE YOU";
50 INPUT AGE
60 IF AGE <= 20 THEN PRINT "MY, HOW YOUNG YOU ARE," ; N$
70 IF AGE > 20 THEN PRINT "WHAT A NICE AGE TO BE," N$
80 PRINT "HAVE A NICE DAY," ; N$
RUN
```

Now try to answer the questions using various names and ages. Also, try to reverse your answers putting your age where your name belongs and vice versa. What happens? The computer will not accept letters where it is expecting numbers. However, it will accept numbers where you might expect it to accept only letters. Why? Because a string is made up of characters. Characters can be either letters or numbers, or for that matter any displayable character available from the ATARI keyboard.

String Comparisons

Just as we have compared two numbers using operators such as <, >, >=, etc., so may we compare strings. In this book, we will delve into the meaning of the equal ("=") and not-equal ("<>") operators as applied to strings.

```
+-----+
| In fact, though, ALL the operators which are |
| available to numbers are also available for |
| use with strings. We would heartily       |
| recommend that you experiment with the other |
| operators. Is "CAT">"cat"? Is "12">"2"?    |
| The answers may surprise you.              |
+-----+
```

Generally speaking, the string comparison operators produce a result by examining each of the two strings being compared a character at a time. That is, the first characters of each string are compared. If they are equal, then the second characters are compared. If they are equal, then the third characters are compared. And so on.

The character-by-character comparison continues until the end of one or both strings is reached. Two strings are equal if and only if they are exactly the same length and if each pair of characters match identically. They are always unequal if their lengths differ.

```
+-----+
| For other comparisons, the first pair of    |
| characters which differ determine the truth |
| of the comparison. If one string is shorter |
| than the other but otherwise matches       |
| exactly, it is considered to be "less" than |
| the longer string.                         |
+-----+
```

The implication of this is that "CAT" is equal to "CAT" and only to "CAT". Of course, you can also use string variables in comparisons, so in this statement

IF X\$="CAT" THEN PRINT "THEY MATCH"
the message will be printed if and only if X\$ has a
LENGTH of 3 and contains the characters "CAT".

Since human responses to computer questions are often
unpredictable, a program such as the following will
often NOT work:

```
10 PRINT "Who is buried in Grant's tomb";
20 INPUT NAME$
30 IF NAME$="GRANT" THEN PRINT "YOU BET YOUR LIFE!"
```

What happens if the person answers "grant" or "Grant" or
"U.S.Grant" or any of several equally valid responses?
Your program counts them as wrong, where a human would
count such variations as correct.

A better choice in such situations is usually the
multiple choice. And here BASIC XL provides a mechanism
to ensure "correct" responses. Recall that we said
above that BASIC XL will only assign as many characters
to a string as the string is dimensioned for. So what
happens if you dimension a string to contain only one
character? Try the following program fragment:

```
10 DIM CHOICE$(1)
20 PRINT "WHO IS BURIED IN GRANT'S TOMB?"
30 PRINT "A. GRANT"
40 PRINT "B. GROUCHO"
50 PRINT "C. LINCOLN"
60 PRINT "YOUR CHOICE (A, B, OR C) ";
70 INPUT CHOICE$
80 IF CHOICE$="A" THEN PRINT "ok!" : END
90 IF CHOICE$="B" THEN PRINT "Wow" : END
100 IF CHOICE$="C" THEN PRINT "Try again":GOTO 60
110 PRINT "you must choose A, B, or C":GOTO 60
```

Do you see the difference? Even if the user answers
"AARDVARKS ARE BEAUTIFUL" in response to your INPUT
prompt, BASIC XL will only store the first letter ("A")
in the variable CHOICE\$, since it was dimensioned to
hold only one character. Sometimes, restricting the
size of a string in this way can make writing string
comparisons much easier.

Exercises:

-
- 1) Write a program to produce a form invitation in which
a name is requested and placed in a string variable.
When the invitation is printed out, including at least
two usages of the name contained in the string variable.
 - 2) Ask the user a question which should be answered with
YES or NO. Have your program accept any answer which
begins with "Y" or "N". Respond with some appropriate
message(s).

CHAPTER XXI

STRING VARIABLES, PART II: EVEN WORDIER

Glossary:

Destination String Is a string to which a value is being assigned.

Source String Is a string whose value was previously stored in the computer's memory.

Substring Is a piece or part of a string's value.

Subscripts Are numerical expressions which specify the beginning and (if 2 subscripts are given) ending character positions within a string variable, thus defining a substring.

LEN() Is a special BASIC function that indicates the number of characters (including blank spaces and graphic characters in a string.

In the previous chapter we discussed string variables and their use with LET and INPUT statements. There are many other advanced programming techniques for handling string variables. In this chapter we will discuss destination strings, source strings, substrings, subscripts, and the LEN function. Mastering these terms and their usage should provide you, as a beginning programmer, with enough latitude to create interesting programs without overwhelming you with other complicated functions.

The examples of LET and INPUT used in our previous discussion of string variables really discussed destination strings. A destination string is one to which a value is assigned. By DIMensioning a string and then assigning a value to it, we use it as a destination string.

With an INPUT statement, the keyboard provides the assignment of the value of the destination string. A destination string is one which is being changed in some manner.

Conversely, a source string is one whose value has been previously stored in the computer's memory. Thus, a source string is actually a string which was previously used as a destination string (i.e., it was DIMensioned and assigned to earlier in the program).

More information concerning source strings will follow throughout this chapter. Try to keep in the mind the differences between destination and source strings.

SUBSTRINGS

With the string variables previously employed, the entire string was accessed. We had no method for accessing and using part of the string. A substring is a division or a part of a string. In order to examine and/or manipulate a substring, you must define the substring by using a subscript.

A subscript consists of one or two numbers, and it must directly follow the string variable's name. The number(s) refer(s) to the characters in a string variable. Examples of subscripts include B\$(20), C\$(1,15), GUESS\$(50), and STATE\$(5,20).

Single Subscripts:

If only one number is present in the subscript, the computer will access the substring beginning with the character in the specified, numbered position. It will continue to access characters until it reaches the end of the string.

Instruction:

You type:

Computer Responds

```
10 DIM STATE$(100)
20 LET STATE$="NEW YORK, NEW JERSEY,
   NEW HAMPSHIRE, NEW MEXICO"
30 PRINT STATE$(5)
RUN
```

```
YORK, NEW JERSEY,
NEW HAMPSHIRE,
NEW MEXICO
```

In the example above, the first statement at line 30 caused the computer to find the fifth character. Beginning from that point, it printed the rest of the string. Now change line 30.

You type:

Computer Responds

30 PRINT STATE\$(15)

RUN

JERSEY, NEW
HAMPSHIRE, NEW
MEXICO

Again, the computer searches for the fifteenth character and executes the command. It PRINTS beginning with the fifteenth character and ends with the last character in the string.

Now change line 30 again.

You type:

Computer Responds

30 PRINT STATE\$(25)

RUN

W HAMPSHIRE, NEW
MEXICO

Notice that each character--letter, punctuation mark, blank space, etc.-- is assigned a position. If a blank space is in the position that the computer is searching for, the computer will PRINT a blank space. It will then use (or continue to use) the rest of the string. Change line 30 again.

You type:

Computer Responds

30 PRINT STATE\$(37)

LIST

RUN

NEW MEXICO

Notice that NEW MEXICO appears to be indented one space. What really happened was we asked the computer to PRINT the 37th character. In this example, a blank space is in position 37. Therefore, the computer PRINTED a blank space.

Remember, when only one number is present in the subscript, the computer will begin at that number and will continue to access characters until it reaches the end of the string.

Double Subscripts:

If two numbers are present in the subscript, the substring will contain all characters within the stated numerical bounds, inclusively.

DIET\$(1,30) means the computer will begin to access the substring at the first character and end with the thirtieth inclusively.

ROAD\$(15,35) means the computer will begin to access the substring at the fifteenth character and continue until it reaches the thirty-fifth character inclusively.

Let's go back to our example program.

Instruction:

You type:

Computer Responds:

LIST

30 PRINT STATE\$(1,8)
40 PRINT STATE\$(11,20)
50 PRINT STATE\$(23,35)
60 PRINT STATE\$(38)
RUN

NEW YORK
NEW JERSEY
NEW HAMPSHIRE
NEW MEXICO

READY

In the first three PRINT statements above, the subscript included two numbers. The computer goes to the particular position designated by the first number in the subscript. It uses that character as a beginning point for the specified operation (in this case, a PRINT operation). It continues to access characters and to perform the operation until it reaches the character numbered the same as the second number in the subscript. It also accesses that last character. (Remember, the subscripted characters are accessed inclusively.)

STRING CONCATENATION

Instruction:

You type:

```
10 A$="HELLO"
20 PRINT "WHAT IS YOUR NAME";
30 INPUT N$
40 A$=A$,N$
50 PRINT A$
```

RUN this program and answer the INPUT prompt. Were you surprised with the results?

The only new thing here is line 40. The operation performed by the comma in this line is called "concatenation". This means that the string following the comma is appended to the string to the left of the comma. Both these strings are right of the equal sign and hence are source strings. What is interesting about line 40 is that A\$ is both a source and a destination string.

There is no real limit to the number of strings which may be concatenated. to illustrate, replace line 40 above with this:

```
40 A$=A$,A$,N$,"AND HI,ALSO."
```

and RUN the resultant program.

Sometimes, you do NOT know the LENGTH of a string variable. For example, in a program which contains an INPUT statement, the DIMension value is usually large enough to allow for all possibilities. How can you determine the actual LENGTH of a string variable? LEN() is a BASIC XL function which will tell you the number of characters including blanks in a string variable. The space in the parentheses is for the name of the string variable.

Instruction:

You type:

```
10 DIM NAMES$(31)
20 PRINT "WHAT IS YOUR NAME (30 CHARACTERS"
25 PRINT MAXIMUM); MAXIMUM)";
30 INPUT NAMES$
```

```

40 IF LEN(NAME$) = 0 THEN GOTO 100
50 IF LEN(NAME$) > 30 THEN GOTO 200
60 PRINT "THANK, "; NAME$:STOP
100 PRINT "DON'T YOU LIKE ME?"
110 PRINT "PLEASE GIVE ME YOUR NAME."
120 PRINT "PRETTY PLEASE?"
130 GOTO 30
200 PRINT "YOUR NAME IS TOO LONG FOR ME"
210 PRINT "TRY TO USE INITIALS OR A NICKNAME"
140 GOTO 20

```

Do you understand this program? Why did we DIM NAME\$ to equal 31 when we asked for a maximum of 30 characters? The reason for DIMensioning NAME\$ to equal 31 characters is rather simple. If we used exactly 30, the computer would reserve only 30 character's worth of space. The computer would chop off anything beyond 30. There would be no way to determine if the string variable NAME\$ contained exactly 30 characters. By allowing for one extra, the computer can make the distinction between exactly 30 and more than 30.

In lines 20 and 25, our PRINT statements indicate the information we want. Notice that the semicolon at the end of the line forces the question mark produced by our INPUT statement to be PRINTed on the same line as the PRINT statement. If the semicolon is not present, the program itself is not affected. The semicolon allows for a neater appearance.

By using an IF...THEN GOTO statement in lines 40 and 50, we have forced the computer to make a decision. First, it must determine the number of character positions in our string variable, NAME\$. The LEN function enables the computer to determine the number of positions. The computer must decide if LEN(NAME\$) is equal to 0. If it is, then the computer will drop to line 100 and execute that line and the ones that follow until it reaches the next GOTO statement.

If LEN(NAME\$) is greater than 30, the computer will GOTO 200 and PRINT lines 200 and 210 before it loops back to line 20.

On the other hand, if LEN(NAME\$) is equal to 30 or less characters, the computer PRINTS line 60 and then STOPS.

We believe that learning to manipulate strings variables is very important. That is why we have devoted so much space to this chapter. Before we have you create a program which examines string variables, we have one more example.

Instruction:

You Type:

```
10 DIM NAMES$(100)
20 PRINT "TYPE YOUR FIRST NAME AND LAST NAME"
25 PRINT "LEAVING ONE SPACE BETWEEN THEM"
30 INPUT NAMES$
40 FOR PTR = 1 TO LEN(NAMES$)
50 IF NAMES$(PTR,PTR) = " " THEN GOTO 90
60 NEXT PTR
70 PRINT "ONLY ONE NAME?"
80 GOTO 20
90 PRINT NAMES$(PTR + 1)
95 PRINT NAMES$(1,PTR-1)
100 GOTO 20
```

line 10--Our program begins by DIMensioning our string variable NAMES\$. Notice, we have told the computer that NAMES\$ might be as large as 100 characters.

lines 20 & 25-- Next, our PRINT statements indicate the information we desire.

line 30-- The INPUT statement provides an opportunity for the user to type into the computer the desired information.

line 40-- We set up a new variable called PTR (pointer). You will recall in Chapter XVIII we used our FOR...NEXT loop as an automatic counter. We are using it here again in that same manner. In line 40 we initialized the variable at one to continue for the LENgth of our string variable NAMES\$. Because the user might enter any name with any number of characters to a maximum of 100, we have no way to determine the exact ending character position in NAMES\$ unless we use the LEN (NAME\$) function.

Remember a string variable with a subscript identifies a character position in the string. Since we have initialized our variable in line 40 to 1, the first time through NAMES\$(1,1) or the first character position. If that position does not contain a blank space, the computer drops to line 60.

line 50--Each time the computer reaches line 50 it tests to see if NAME\$(PRT,PRT) is equal to a blank space. The computer continues in the FOR...NEXT loop until it finds the blank space.

line 60-- Contains the NEXT statement. The value of the variable PTR increases by 1. In essence, this means $PTR = PTR + 1$ GOTO 40. The computer then determines if the new value of PTR is still within the range of 1 to LEN(NAME\$).

line 70--This PRINT statement was included in case the user entered only one name.

line 80--Our GOTO statement sends the computer back to line 20 to get the user's first and last name separated by a blank space.

line 90-- The computer executes this line when it finds the blank space. Line 90 means PRINT our string variable NAME\$ from the character position which is equal to the blank space plus 1. For example, if the user's name was WOLFGANG MOZART, the FOR...NEXT loop would begin with the W. It would determine that W is not a blank space and thus it would move to the next PTR character position, 0. The procedure would continue until the blank space is found.

Count the character positions in WOLFGANG MOZART. In what character position is the blank space? The blank is in the ninth position. Therefore, in line 90 $PTR + 1$ is equal to 10. The letter in the tenth position is M. When a variable has only one subscript number, the computer will access all characters beginning with that number (10 in our example) and PRINT until it comes to the last character in the string. Because subscripts are inclusive, the last character is also PRINTED. Up to this point the computer would PRINT MOZART.

line 95--Adjacent to the blank space, we tell the computer to PRINT NAME\$(1,PTR-1). This means PRINT our variable NAME\$ beginning with the first position and continuing until the position of PTR-1 is reached. In our WOLFGANG MOZART example, PTR was position nine so PTR-1 is equal to position 8. When a string variable is followed by a subscript containing two numbers, the computer uses the first number as a beginning point and the second as an ending. Our program produces MOZART WOLFGANG, or any name which was properly entered. Each time a name is entered the results will be last name and first name on separate lines.

If you have trouble understanding this program, play computer. Using paper and pencil, go through each step just the way the computer would do it. That should make the explanation easier for you to understand.

Also please note: The above program is not perfect. We are aware of the flaws. By placing a space before the first name, a user could cause problems with this program. However, we are using it to explain string variables. If you noticed the flaws, that's great.

```
SPECIAL NOTE: Actually BASIC XL
gives an easier way to find the
space in NAME$ if it exists. Try
replacing lines 40 through 60 with
this:
```

```
40 PTR=INSTR(NAME$, " ",0)
50 IF PTR THEN GOTO 90
(DELETE LINE 60)
```

```
For more information on INSTR, see
your BASIC XL Reference Manual.
```

Exercise:

-
- 1) Write a program to permit INPUT of a name and then PRINT the name backwards.

Chapter XXII

SOUND: YOU CALL THAT MUSIC

Glossary:

SOUND -----	A BASIC statement which tells the computer to generate a tone or noise
Sound Channels -----	Are numbered from 0 through 3 and each channel represents a separate tone or noise.
Pitch -----	Is the depth of a tone or of a sound. The range of pitch is numbered from 0 through 255.
Sound Quality -----	Can be selected values between 0 and 15 and allows the ATARI to produce pure or distorted notes or silence.
Volume -----	The degree of loudness or audibility, represented by a number from 0 to 15.

The ATARI'S capacity to produce sound provides another dimension for your programming creativity. With elementary programming skills, you can force the computer to execute commands and produce sounds which would require advanced skills on some other home computers.

In order to make a sound, you must first indicate to the computer that you want a sound. The SOUND statement tells the computer to generate a tone. A SOUND statement is followed by four numbers. Each number is separated from the others by commas.

Example:

10 SOUND 1, 15, 10, 5

The first number following SOUND indicates the sound channel. Your ATARI gives you a choice of four channels numbered from 0 through 3. Each channel is independent of the others. Because of this, it is possible to blend the sound channels together. This blending of sound would be similar to voices in a chorus or musical instruments in an orchestra. Each sound channel is controlled by a separate SOUND statement.

Following the sound channel in a SOUND statement is the pitch. The ATARI is capable of producing 256 different pitches, numbered 0 to 255, but not all pitches produce sounds on all modes of "quality".

The third number in a SOUND statement regulates the sound quality. With a value between 0 and 15, the sound quality is capable of producing pure tones or of producing tones mixed with different amounts of noise. Pure tones are generated by the values 10 and 14. These tones are similar to those produced by a flute. Value 12 is also useful when programming music. It simulates a reed sound similar to a clarinet.

Other even numbered values introduce various amounts of noise into the pure tone. These values are useful for creating sounds like car crashes, explosions, and video games. Odd-numbered distortion values (1,3,5,7,9,11, and 13) produce silence.

SPECIAL NOTE: When the sound quality is 10 or 14, the pitches generated increase their frequency regularly with 255 producing the lowest pitch. With a range of one octave below and two octaves above middle C, the ATARI can produce all notes: sharps, flats, and naturals. In addition there are other tones available that do not correspond to notes on the musical scale. Although they are of no value in programming music, they can be used as alarms or sound effects to accompany other programs.

Instruction:

You type:

```
10 FOR FREQ = 10 TO 200 STEP 10
20 FOR VOL=15 TO 0 STEP -1
30 SOUND 0,FREQ,10, VOL
40 NEXT VOL
50 NEXT FREQ
```

What did you hear? Did you notice the variations in sound?

In line 10 of the above program, we used the variable FREQ for determining the pitch. The range of values for FREQ were selected to be between 10 and 200. In addition, we selected STEP 10 to increase the pitch. By using STEP 10, we made the variations in pitch more apparent than it would have been had we selected STEP 2 or STEP 4, etc.

We also chose to decrease the volume. In line 20 we set the range of values for the volume to decrease 15 to 0, STEP-1. Remember, variables may decrease as well as increase.

Lines 40 and 50 simply close our FOR and NEXT loops. This allows the computer to automatically make the changes in pitch and volume.

Working with the sound element of your Atari requires practice. If you are musically inclined, the differences in pitch and sound quality will be more apparent to you than to someone with a less sensitive ear.

Even if you cannot distinguish small differences in sounds, you can still create interesting tones to accompany your programs.

Actually, the Atari permits such a wide range of sounds that your imagination and creativity should be your only limitation.

The following program simulates a space ship taking off.

Instruction:

You type:

```
10 FOR P=150 TO 100 STEP -1
20 FOR N=P TO P-100 STEP-5
30 SOUND 0,N,10,5
40 NEXT N
50 NEXT P
60 GOTO 10
```

In order to create the whirling sound of the space ship ascending, it was necessary to form two FOR...NEXT loops. In line 10 we establish the first loop. Our variable P has a range of 150 to 100, and it will decrease one number each time the loop is executed.

Line 20 starts our second FOR...NEXT loop. A new variable N is initialized to have the same value as P. Each time through the loop, the value of N will decrease by 5 until it reaches the value of P-100.

In order to produce noise, you must use a SOUND statement, like the one shown in line 30. The first number indicates the sound channel, in this case 0. The next number is our variable N, which indicates the pitch. The third number is sound quality; in this case we used 10 which produces a pure tone. Finally, the last number indicates volume.

In this example, the FOR...NEXT loops are nested. That is the FOR N loop is inside the FOR P loop. Remember, the last FOR's variable name gets the first NEXT's variable name. In lines 40 and 50 we complete the loops we began in lines 10 and 20. Notice that the value of N will change more often than the value of P. The innermost FOR...NEXT loop must be completed before the value of P is changed.

Let's play computer to see how this program would actually work. The first time through the program the value of P would equal 150 and the value of N would equal 150. When the computer reaches the SOUND statement in line 30, it replaces N with the value of 150. When the computer reaches line 40, it goes back to line 20 and changes the value of N to decrease by 5. N then has the value 145. The value of N will continue to decrease by 5 until it reaches a value of P-100 or 50. Once the value of N equals 50, the computer will drop down to line 50. Line 50 will cause the computer to return to line 10. The value of P will now decrease by 1, and the value of P will equal 149. When the computer drops down to line 20, the value of N is now equal to 149. Again, the computer will repeat the FOR N loop until the value of N is equal to P-100. The value of N would now equal 49.

By using the GOTO statement in line 60, we create an endless loop which will cause the computer to continue to return to line 10. Because line 10 contains the FOR P loop, the value of P is reinitialized each time the computer executes line 60.

Exercises:

- 1) Write a program which creates random sounds.
- 2) Write a program which creates random pure tones.

Chapter XXIII

GRAPHICS PART II: I CAN WRITE BIGGER THAN YOU

Glossary:

Graphics Window	The large area of the monitor or television screen in which graphics words, designs, or pictures can be displayed.
Text Window	An area at the bottom of the monitor or television screen which contains enough area for four lines of text.
PRINT # 6;	A command used in GRAPHICS modes 1 and 2 that causes the characters to appear in the graphics window.
POSITION	A statement which moves the cursor to any specified location on the screen. The location is designated by the column number and the row number.

As stated earlier, the ATARI Home Computer is equipped with a large range of graphic features. Chapter XIV was intended to introduce you to the most common of those aspects. This chapter and the ones that follow will further explain the ATARI'S graphic capabilities and the concepts necessary to create your own advanced designs.

Like GRAPHICS 0, GRAPHICS 1 and GRAPHICS 2 are referred to as the "text modes". They are used to display words and characters. The difference is in the size of the text. Think of GRAPHICS 0 as small, GRAPHICS 1 as medium, and GRAPHICS 2 as large.

Enter the new modes by typing GRAPHICS 1 or GRAPHICS 2. Press [RETURN]. Notice that the display screen is now split into two segments. The first segment occupies about five/sixths of the screen and is referred to as the graphics window. The second section has enough area to hold four lines of text and is called the text window.

Instruction:

You Type:

```
GRAPHICS 1
PRINT "VARIETY IS THE SPICE OF LIFE"
PRESS {RETURN}
```

Your window text screen should look like this:

VARIETY IS THE SPICE OF LIFE
READY

These instructions allowed you to PRINT a line of normal size text along the top of the text window. In order to increase the size of the characters and to have them displayed in the larger segment of the screen or the graphics window, you must use a PRINT #6; statement.

Now type:

PRINT #6; "VARIETY IS THE SPICE OF LIFE"

Notice the size of the characters in the graphics window. To see even larger characters, change to GRAPHICS 2.

You Type:

GRAPHICS 2

PRINT #6; "VARIETY IS THE SPICE OF LIFE"

Again, you will see the characters are now even larger than they were in GRAPHICS 1.

In both of our examples, the characters appeared in the upper left hand corner of the screen. For creative purposes this placement may not suit your needs. In order to change the location of the characters, a POSITION statement is used.

As you will recall in our discussion of GRAPHICS 0 we compared the display screen to a piece of graph paper. In GRAPHICS 1, the larger characters limit the screen size to 20 columns by 20 rows. (Remember rows run horizontally while columns run vertically.) Because of the even larger sized characters in GRAPHICS 2, there is room for 20 columns and only 10 rows.

Now you have the amount of area in which you can POSITION the characters. A POSITION statement is used in conjunction with column and row numbers. The POSITION statement does not visibly move the cursor location when it is executed. It does move, however, when a subsequent statement such as PRINT accesses the display screen.

You Type:

```
GRAPHICS 1
POSITION 7,11
PRINT #6,"MIDDLE"
```

These instructions should place the word "middle" in the center of the graphics window.

There are other variations you can use with GRAPHICS 1 and 2. In the text window, lower-case and/or inverse video characters as well as upper-case may be used. When these same characters are used in the graphics window, however, they translate into color variations. In the exercise below, first type A, the lower case A, control A, inverse upper case A, inverse lower case A. (Remember the ATARI key produces an inverse image.)

Instructions:

You type:

```
GRAPHICS 1
PRINT #6; "AaAAa"
```

What did you see? Notice all the variations of color available in GRAPHIC 1 and 2.

GRAPHICS modes 1 and 2 give your creative urges an additional outlet. Although all our examples in this chapter have been in direct mode, it is also possible to use programming mode.

```
10 GRAPHICS 2
20 PRINT #6; "WHAT a difference this MAKES"
30 PRINT #6; "IT'S ALL RIGHT TO HAVE A SPLIT PERSONALITY"
```

In our program above, some information is displayed in the graphics window and some appears in the text window.

Exercises:

1) Write a program which will PRINT in the graphics window a person's name and "YOU ARE A STAR"

2) Write a program using the multiplication sign that will look like a marquee. PRINT your name in the middle of the marquee and center the marquee on the screen.

Chapter XXIV

GRAPHICS PART III: MY PICTURE IS IMPROVING

Glossary:

Graphic Mode -----	A state in which the computer responds to instructions for the purposes of drawing pictures, designs, graphs, or variations of the standard characters.
Graphics Window -----	The large area of the monitor or television screen in which graphic words, designs, or pictures can be displayed.
Text Window -----	An area at the bottom of the monitor or television screen which contains enough area for four lines of text.
Pixels -----	The shaded blocks of color used in graphic modes number 3 and above to create pictures, designs, and graphs.
COLOR -----	A statement which selects one of the available color registers to be used with subsequent PLOTs and DRAWTOs.
PLOT -----	A statement which illuminates a single point on the screen.
DRAWTO -----	A statement which causes a line to be drawn from a point (the last plotted point) to a specified location.
LOCATE -----	A statement which allows a program to find out what color is already plotted at any given point on the screen.

This chapter discusses the GRAPHICS statement in two parts. First, we discuss graphic modes 3, 5, and 7. Although GRAPHICS 7 was previously described in Chapter XIV, it is worth repeating here for comparison to modes 3 and 5 and for the sake of completeness. Second, graphics modes 4 and 6 are introduced a little later because of differences between them and the other modes.

Pixels in Modes 3, 5, and 7

The big difference between modes 3, 5, and 7 when compared to modes 0, 1 and 2 is that we are now dealing with pixels instead of characters. Recall that pixels

are shaded blocks of color. Earlier we described the screen in graphics mode as being similar to a piece of graph paper. We noted that a pixel is equivalent to one block being filled with color.

In addition, graphics modes 3, 5, and 7 differ from each other in the number of pixels available. In general, the higher the number in graphics mode, the more pixels or shaded squares are available. In terms of picture quality GRAPHICS 7 provides finer lines than GRAPHICS 3. GRAPHICS 7 has better resolution, but GRAPHICS 3 makes it easier to produce bar graphs. Each graphic mode has its own special abilities. As you become more familiar with the various modes, you will have an easier time choosing which mode best suits your needs.

The chart below summarizes the pixel size characteristics of each of the five modes discussed in this chapter (modes 4 and 6 are included here for convenience, even though they are not discussed until later).

Graphics Mode	Number of Rows	Number of Columns	Total # of Pixels
3	20	40	800
4	40	80	3200
5	40	80	3200
6	80	160	12800
7	80	160	12800

Pictures in Modes 3, 5, and 7

The choices of colors available in graphics modes 3, 5, and 7 are the same. These colors are as follows:

COLOR 0 = black
 COLOR 1 = orange
 COLOR 2 = aqua
 COLOR 3 = blue

The color black is used for background.

When using GRAPHICS modes 3 through 7, PLOT and DRAWTO statements are usually used. As stated in Chapter XIV, the PLOT statement enables the user to select a particular point position. The first number after PLOT tells the computer the column desired; the second number indicates the desired row. Numbers in a PLOT statement are separated by a comma.

When PLOT and DRAWTO are used in pairs, the PLOT statement indicates the starting point of a line while the DRAWTO statement indicates the ending point. As with PLOT, when you use DRAWTO you must specify two numbers (separated by a comma) to indicate the desired end position as a column and row position.

When a DRAWTO follows a previous DRAWTO without an intervening PLOT, the ending position specified by the first DRAWTO becomes the starting position for the second one.

(If you need more information concerning PLOT and DRAWTO, refer to Chapter XIII and/or your BASIC XL Reference Manual.)

Instruction:

You type:

```
NEW
10 GRAPHICS 3
20 COLOR 1
30 PLOT 0,0
40 DRAWTO 39,0
50 COLOR 2
60 DRAWTO 39,19
70 COLOR 3
80 DRAWTO 0, 19
90 COLOR 0
100 DRAWTO 0,0
RUN
```

The program above gives directions for drawing a rectangle. Was the image on your screen a three-sided or a four-sided object? Actually, all four sides were drawn. Why did the fourth side appear to be missing? In GRAPHICS 3, 5, and 7 there are three available foreground colors. The fourth color (COLOR 0) is always the background color and so is indistinguishable from the background when it is plotted or drawn.

Notice also how much screen area is covered by our program. Try to keep that image in your mind.

Instruction:

You Type:

```
10 GRAPHICS 5 {RETURN}
We are going to change graphics modes.
```

In order to do that, we need only change line 10. Remember, the computer disregards our previous line 10 and keeps the rest of the program intact.

Instruction:

You Type:

RUN

How does our object change? It becomes smaller. Why? In GRAPHICS 3, the individual pixels covered a large area; in GRAPHICS 5 there are more pixels available, but each individual pixel is smaller.

Instruction:

You Type:

10 GRAPHICS 7 {RETURN}
RUN

Again our object became smaller. The lines of color created in GRAPHICS 7 are finer than those in GRAPHICS 3. However, in GRAPHICS 7 more lines would have to be drawn to occupy the area of one line in GRAPHICS 3. GRAPHICS 3 is similiar to drawing with a magic marker while GRAPHICS 7 represents a fine point pen. GRAPHICS 5 fits in the middle like a felt tip pen.

Graphics Modes 4 and 6

Graphics modes 4 and 6 have the same screen size as graphics modes 5 and 7, respectively (see our chart above). The big difference between these modes is in the number of available colors. GRAPHICS 4 and 6 have only two available colors. One color is used for foreground and one is used for background.

Instruction:

You Type:

10 GRAPHICS 4
RUN

What happened to the object? How many sides can you

see? Although our original program specifies four colors, only two are available in GRAPHICS 4. Each time the computer encounters an even numbered color, it uses the available foreground color; odd numbered colors become black.

Instruction:

You Type:

10 GRAPHICS 6

RUN

You will notice a smaller sized object in one color. Because of the limited number of colors, GRAPHICS 4 and GRAPHICS 6 are rarely used.

The LOCATE Statement

When we have a picture in mind which we want to draw on the screen, we usually use COLOR, PLOT, and DRAWTO to accomplish our ends. But suppose there is a picture already on the screen and we want to "describe" it to the computer. Of course, one way would be to get a magnifying glass and start typing in pixel locations and colors in response to INPUT statements, but this is obviously ridiculously slow, tedious, and error-prone.

Fortunately, BASIC XL provides a method--the LOCATE statement--by which the computer can directly "read" the screen itself. The form of this statement is:

LOCATE column, row, color

and examples of use might include:

LOCATE X,Y,C

LOCATE HORIZONTAL/2+10,VERTICAL/2+5,LOCCOLOR

While the column and row may be specified by arithmetic expressions (just as with PLOT and DRAWTO), the "color" MUST be an arithmetic variable. When a LOCATE statement is executed, BASIC XL causes the computer to "read" the pixel located at the specified column and row. It places the value it reads into the variable which you supply. And, not surprisingly, the data read is actually the color of the screen at that point. Note that this color value is the same value used in the COLOR statement which is then subsequently placed on the screen by PLOT or DRAWTO.

Instruction:

You type:

```
10 GRAPHICS 3
20   FOR COUNT=1 TO 400
30   COLOR RANDOM(1,3)
40   PLOT RANDOM(40),RANDOM(20)
50   NEXT COUNT
60 PRINT "NOW CONVERTING"
70   FOR COLUMN=0 TO 39
80     FOR ROW=0 TO 19
90       LOCATE COLUMN,ROW,SCREENCOLOR
100      COLOR 2
110      IF SCREENCOLOR=0 THEN PLOT COLUMN,ROW
120      NEXT ROW
130    NEXT COLUMN
140 PRINT "FINISHED"
```

Let's analyze this program by line number.

10--We select the largest pixels available, to make our program more visible.

20 to 50--We simply plot a lot of points on the screen using randomly chosen colors and locations.

60--Just a message (which will be displayed in the text window) to tell us when the program gets to this point.

70 and 130--A loop, whereby we check all the columns on the screen.

80 and 120--Another loop, but here we check all the rows within each of the columns.

90--We check a particular location to find out what color is on the screen.

100 and 110--If the location we checked had not previously been plotted, we change it to COLOR 2 via the PLOT statement.

140. We're done.

The effect of this program is to sprinkle various colors on the screen. Then we go back and change all the "background" pixels to another color. A relatively useless program, but it illustrates the principles of PLOT, COLOR, and LOCATE well.

Exercises:

1) Modify the exercise at the end of Chapter XIV so that the programs RUN in GRAPHICS 3 and then in GRAPHICS 5.

2) Modify lines 100 and 110 of the last example above to change the color of EVERY pixel on the screen. The new colors should depend on the old colors as follows:

old color	new color
0	3
1	2
2	1
3	0

Hint: Don't use an IF statement to determine the new color. Work out an arithmetic expression instead.

Chapter XXV

GRAPHICS PART IV:

ALL THE COLORS OF THE RAINBOW

Glossary:

Color The memory locations which set the colors used
Registers for foreground, background, and border colors.

COLOR A statement which selects one of the available
----- color registers and with that color draws.

SETCOLOR A statement which allows the user to change
----- colors on the screen at any time.

Luminance The brightness or the radiation or reflection
----- of light.

Hue The color or the particular shade or tint of a
---- color.

For many people, this chapter will prove difficult and confusing. Try to stick with us through the charts, explanations, and examples. We believe that it will become clear in the end.

In our previous chapters on graphics we have discussed the COLOR statement. In review, we said a COLOR statement is one which selects an available color number and that subsequent PLOTs and DRAWTOs use that color number to place pixels on the screen. For example, in modes 3, 5, and 7, it was possible to select three foreground colors and one background color for PLOT and DRAWTO statements.

But the Atari Computer and BASIC XL are capable of much more complex operations. To try to find a real word analogy, let's think of a COLOR number as selecting a particular spot on an artist's palette, regardless of what shade of paint is on the spot. Well, just as an artist might not place the same color on the same spot on his palette each time he paints, so does BASIC XL (or, more properly, the Atari Computer's hardware) change which shade a particular COLOR number refers to.

To continue our analogy, we might say that the computer simply changes which jar of paint it dumps onto each spot. The situation gets more complex: you, as the programmer, can change the color of the paint in the jar

(by using the SETCOLOR statement). And, just as the COLOR statement is not really aware of what shade is actually "painting", so is the computer not really aware of what shade is in the jar.

The net result of all this is that you really do have control of what color goes where on the screen. The only problem is that you have to follow the rules about what COLOR number refers to what palette spot (which can vary depending on the GRAPHICS mode chosen) and what SETCOLOR refers the "jar" which is used for that palette spot. All of the options for graphics modes 3 through 8 are presented in table form below (Figure 2). But before we discuss the table, we will investigate the workings of SETCOLOR, the statement which lets us change the colors in the jars.

The SETCOLOR Statement

Following SETCOLOR are numeric expressions. Separated by commas, these numbers determine which color register to change, to which hue to change it, and what degree of brightness the hue will be.

The first number following SETCOLOR indicates which register to set. The registers are numbered from 0 to 4 and relate to the jars of paint in our analogy. To find out which palette spot they refer to you must consult Figure 2, below.

The second number after SETCOLOR determines the hue. The sixteen hues are numbered 0 to 15 and are usually visiblek as follows, though there can be variation from one television screen or monitor to another.

0 Grey	8 Light Blue
1 Gold	9 Blue-Green
2 Orange	10 Aqua
3 Red	11 Green-Blue
4 Pink	12 Green
5 Violet	13 Yellow-Green
6 Blue-Purple	14 Orange-Green
7 Blue	15 Orange

Figure 1: Available Hues

The last number following SETCOLOR represents the brightness or luminance. Although numbered from 0 to 15, the even-numbered values are the only settings which are meaningful. The 0 represents the darkest value and 14 represents the brightest value. (Odd numbers are treated the same as the next lower even number.)

Instruction:

You Type:

```
100 GRAPHICS 3
110 COLOR 1
120 PLOT 0,10
130 DRAWTO 39, 10
140   FOR HUE=0 TO 15
150     FOR LUM=0 to 14 STEP 2
160       PRINT HUE, LUM
170       SETCOLOR 0, HUE, LUM
180       FOR WAIT=1 TO 500
190         NEXT WAIT
200       NEXT LUM
210     NEXT HUE
220 END
RUN
```

If you have a color monitor or television screen for your ATARI home computer, the above program will produce each of the sixteen colors available and in addition will produce eight different degrees of luminance or brightness.

Let's analyze the structure of this program. To begin with we must choose a graphics mode. Graphics 3 was selected for two reasons: 1) because it's degree of resolution allows for excellent bars to be drawn; and 2) because it contains many color choices.

Next, we must PLOT a point and also use the DRAWTO statement. These two statements, used in conjunction with COLOR, select a point and create a bar of color. We could have used only one or two points. However, it would almost be impossible to see the changes in the color and luminance.

In line 140 we have begun a FOR...NEXT loop. Our variable is HUE and represents the various color selections available. We could have named the variable almost anything, but HUE is more meaningful than just a letter or another word. Also, note the value of our variable HUE is equal to the sixteen available color choices.

Again, in line 150, we have begun a FOR...NEXT loop. This time our variable is named LUM and represents luminance or the various degrees of brightness available. Earlier we said that only the even values of luminance were meaningful. In our program, the value of our variable LUM is equal to the degrees of brightness available. We have added STEP 2 to our FOR statement so that the value of LUM will increase in even increments.

In order to see which color and which degree of brightness is currently on the screen, we have included the PRINT statement in line 160. Finally, in line 170 we get to the point of all of this. Here is our SETCOLOR statement. As the values of our variables HUE and LUM change, so will the colors of the bar on the screen. This is possible because of our SETCOLOR statement. Notice, though, that the SETCOLOR uses color register 0 while our PLOT statement used COLOR 1. How come? Again, it all has to do with palette "spots", paint jars, etc. Be sure and see Figure 2, below, for a clarification of these confusing numbers.

In line 180 we begin yet another FOR loop. We called our variable WAIT and made its value equal to from one to 500. Although the program will display a bar of color and the various changes in HUE and LUM, without this FOR...NEXT loop, these changes would occur very rapidly. By adding a FOR...NEXT loop whose only function is to keep the image on the screen, we allow the user an opportunity to view the bar of color and its changes for a reasonable amount of time. We could change the value of our variable WAIT depending on our desire to lengthen or shorten the amount of time the various colors are shown.

Whenever you use more than one FOR...NEXT, you must nest them. That is, you put one inside of another, like a nest of tables or a set of mixing bowls. Although you may use as many FOR...NEXT loops as you desire, they must be built from the inside out. The last FOR gets the first NEXT.

```
FOR HUE
  FOR LUM
    FOR WAIT
      NEXT WAIT
    NEXT LUM
  NEXT HUE
```

In lines 190, 200, and 210, we complete all of our FOR...NEXT loops. Our automatic counters are complete and the values on our variables automatically change. Thus, we have the changes in color and brightness.

The Relationship of COLOR to SETCOLOR

Finally, we present the long-awaited Figure 2, which shows in tabular form the relationship between COLOR number and SETCOLOR register. We will present the figure first and then explain it.

Graphics Mode	Color Number	Setcolor Register	Default Hue,Lum	Default Color
3, 5, 7	0	4	0,0	Black
	1	0	2,8	Orange
	2	1	12,10	Green
	3	2	4,6	Dark Blue
4, 6	0	4	0,0	Black
	1	0	2,8	Orange
8, 0	0	2	4,6	Dark Blue
	1	1*	4,10*	Light Blue

(* In mode 8, the hue of SETCOLOR register 1 is NOT selectable-- it will always be the same as the hue or register 2.)

Figure 2: COLOR/SETCOLOR Table

To use this chart, follow these steps.

1. Determine the GRAPHICS mode you are going to use. Use only the portion of the table which applies to that mode.
2. If the colors given in the rightmost column are adequate to your needs, simply use the digit from the Color Number column in your program (in conjunction with COLOR, of course).
3. If you like some of the colors, reserve their Color Numbers for use in your program.
4. Use SETCOLOR to choose a new color(s) by varying the hue and luminance to your choice(s) using any of the Setcolor Registers which are available in your chosen graphics mode. Then use the Color Number which relates to that register in your program.

Instruction:

You Type:

```
10 GRAPHICS 3
15 LET LCOLOR=1
20 FOR HORIZ=0 TO 39
30 COLOR=LCOLOR
40 PLOT HORIZ,0
50 DRAWTO HORIZ,23
60 LET LCOLOR=LCOLOR+1
70 IF LCOLOR>3 THEN LCOLOR=1
80 NEXT HORIZ
100 LET BAR1=4
110 LET BAR2=8
120 LET BAR3=12
200 SETCOLOR 0,8, BAR1
210 SETCOLOR 1,8, BAR2
220 SETCOLOR 2,8, BAR3
230 LET TEMP=BAR1
240 LET BAR1=BAR2
250 LET BAR2=BAR3
260 LET BAR3=TEMP
300 FOR WAIT=1 TO 50
310 NEXT WAIT
400 GOTO 200
```

RUN

What results do you get from this program? How was the movement created? As with marquee lights, the bars are not moving although they appear to be. The movement is created by changing the colors of the vertical bars.

Because we are using bars, we again use GRAPHICS 3. As you become more familiar with each of the graphics modes, it will become easier for you to determine which mode will best suit your needs.

In line 15 we initialize our variable LCOLOR at 1. Because we are in GRAPHICS 3, we have a choice of three foreground colors. In order to create the movement we will want the colors to change. Therefore, we use a variable.

Next, we begin a FOR...NEXT loop. Since the screen size in GRAPHICS 3 is 39 columns, our automatic counting variable HORIZ is set for 0 to 39. This accounts for all the columns on the screen.

Line 30 represents our COLOR statement. The value of COLOR will be set to the value of our variable LCOLOR. Each time the value of LCOLOR increased, our COLOR statement will change.

In order to create an image, in lines 40 and 50, we PLOT and DRAWTO. Notice that the point that we PLOT is dependent upon the value of our variable HORIZ. Also, you can see by using 0 as the row value, the points PLOTEd will be across the top of the screen. By using the value of the variable HORIZ in the DRAWTO statement, we cause a line to be drawn from each value of HORIZ. In other words, we cause a line to be drawn from the top of each column to the bottom of each row.

We use our non-algebraic expression

```
LET LCOLOR = LCOLOR + 1
```

to change the value of the variable LCOLOR. Each time the computer executes line 60, it increases the value of LCOLOR by 1.

Why did we include line 70? As you may remember, in GRAPHICS 3, there is a choice of three foreground colors and one background color. If we did not include line 70, sometimes (every fourth time), we would get black bars. To avoid using black, we use an IF...THEN statement along with our relational operator > (greater than). When the value of LCOLOR is greater than 3, the computer resets its value to equal 1.

In line 80, we complete our FOR...NEXT loop which was started in line 20. The value of our variable HORIZ changes from 0 to 39 in increments of one.

If you add the line, 90 STOP, you will be able to see the bars of color across the screen. Remember to erase line 90 typing:

```
90 {RETURN}
```

In lines 100, 110, and 120 we are initializing three new variables called BAR 1, BAR 2, BAR 3 respectively. Notice the values chosen for these new variables. Remember in a SETCOLOR statement the last number represents brightness or luminance, and only even values between 0 and 14 are meaningful. The values for the variables BAR 1, BAR 2, and BAR 3 were chosen on that basis. To change colors within the program we use our SETCOLOR statement. The first number represents the color register. The second delineates the color desired; in this case, color 8, light blue. The last number determines brightness which depends on the value of our variables BAR 1, BAR 2, and BAR 3.

In order to get the movement, we are going to change the brightness value. To do this lines 230, 240, 250 and 260 are set up to rotate the luminance value. Each time these lines are executed, their values will change. However, the values will always be either 4, 8, or 12.

Finally, in line 300 and 310 we set up another FOR/NEXT loop. The purpose of this loop is to allow the image to stay on the screen so that the viewer can enjoy the color changes.

In line 400, the GOTO statement continues the process. The bars of color remain the same with the SETCOLOR statements continuously reexecuted (executed again and again).

Exercises:

- 1) Write a program to randomly change the background color for text mode (GRAPHICS 0) continuously.
- 2) Write the same program for a display in GRAPHICS 3.

CHAPTER XXVI

GRAPHICS PART V: A REVOLUTION IN RESOLUTION

Glossary

GRAPHICS 8 -----	A special two color graphics mode with a high level of resolution.
Luminance -----	Brightness or the radiation or reflection of light.
Hue ---	Color or the particular shade or tint of a color.

GRAPHICS 8 is discussed as a separate unit because it is significantly different from the other graphic modes previously covered.

As you might imagine, GRAPHICS 8 provides the highest level of resolution. In split-screen mode-- that is the graphics window with text window--the screen size is 320 rows by 160 columns. (Remember, rows run horizontally and columns run vertically.) GRAPHICS 8 is most useful for detailed drawings and pictures. This mode may also be used for games which require a large area to be used as a playfield. Thus, GRAPHICS 8 is most useful where detailed drawings are important.

The reason for the high level of resolution is due to the size of each pixel. In graphics mode 8, each pixel is smaller than it was in the previously discussed modes. The pixels in this mode look more like points than squares (even though they really are still square). This difference allows for more detailed drawings.

Instruction:

You type:

```
NEW
10  FOR MODE 3 TO 8
20  GRAPHICS MODE
30  PRINT "MODE: "; MODE
40  SETCOLOR 2,8,0
50  COLOR 1
60  DRAWTO 0,19
70  DRAWTO 19,39
80  DRAWTO 0,0
90  FOR I=1 TO 1000: NEXT I
100 NEXT MODE
RUN
```

The above program displays the same triangle in six different modes. Notice that in mode 3 almost the entire screen is filled. As the number of the mode increases, the resolution also increases. The size of the triangle becomes smaller, and the lines are finer and more distinct.

When the program reached GRAPHICS 8, what differences did you notice? First, the lines were very fine, and the triangle as the object was clearer and more distinct. What happened to the split screen mode? Did it seem to disappear? GRAPHICS 8 has special characteristics. Because of these special traits, it seems as though the screen is no longer split.

Unlike other graphics modes, the foreground and background colors are NOT independent. By foreground we mean the area in which you would see the text in GRAPHICS 0 or where you would see the design, picture, or drawing in modes 3 through 7.

In Graphics mode 8, the foreground and background colors are always the same hue. Obviously, this means that if the luminance of the two is the same or similar, it is hard to distinguish one from the other. However, by using luminances which are as different as possible, the result can be quite "distinct".

```
-----
| Note: Actually, the colors and |
| luminances that can be selected in |
| GRAPHICS 8 are identical with the colors |
| and luminances available in mode 0, the |
| primary text mode of the Atari computer. |
| You might try some of the various |
| SETCOLORs we did here in mode 0 as well. |
|-----
```

Instruction:

You type:

```
GRAPHICS 8
PLOT 10,10
DRAWTO 100,100
```

What appeared on your monitor or screen? It should have been a diagonal line. If you have a hard time seeing it, it's not your eyes. The line really does not show up very well.

Instruction:

You type:

SETCOLOR 2,8,0

Can you see the line more clearly now? How did the SETCOLOR statement change the line?

The SETCOLOR statement in GRAPHICS 8 is very important. As you will recall from Chapter XXV, the first number in a SETCOLOR statement fills the color register. In graphics mode 8, the user has a choice of three SETCOLOR registers: 1,2 or 4.

Instruction:

You type:

GRAPHICS 8
PLOT 10,10
DRAWTO 100,100
SETCOLOR 1,8,15

Can you still see the line? With the above SETCOLOR statement, you can see the line, but it is not as dramatically clear as it was in the previous SETCOLOR statement. Why? SETCOLOR 1 affects the foreground luminance only. In this case, the hue is ignored. That means the brightness of the foreground can be altered, but nothing else can be changed.

A SETCOLOR 2 statement affects the background hue and luminance. That means, the background can be made darker or lighter. In our statement, SETCOLOR 2,8,0, we made the background as dark as possible. The line looks brighter and more distinct because the background is darker.

A SETCOLOR 4 affects only the border hue and luminance. Note this is exactly the same manner in which hue and luminance work on characters in graphics mode 0.

Exercise:

1) Write a program to draw a border around the edge of the mode 8 screen.

CHAPTER XXVII

PROGRAMMING AIDS: COSMETIC SURGERY

Glossary:

REM Is a statement which provides information
--- concerning the program in a LISTing.

NUM Is a statement which provides automatic program
---- line numbering.

RENUM Is a statement which automatically rennumbers
----- the program lines of an existing program.

The people who write languages like BASIC XL (systems programmers) understand the many problems beginning programmers may encounter. To help alleviate some of those troublesome areas, they include special features in their languages. BASIC XL has many built-in features. In this chapter we will discuss a few which are particularly useful to the beginning BASIC XL programmer.

Have you ever found a note written to yourself? You look at it and for the life of you, you just cannot imagine what you meant by it? The same thing can happen with programs you write.

You may not have any trouble remembering what a program of less than 10 lines does. For example, do you remember this program from Chapter 10?

```
10 LET I = 1
20 PRINT RANDOM (1,100)
30 LET I = I+1
40 IF I < 7 THEN GOTO 20
```

In case you do not, this program will produce 6 random numbers. A much easier way to remember would be to write the program like this:

```
5 REM THIS PROGRAM PICKS 6 RANDOM NUMBERS
10 LET I = 1
20 PRINT RANDOM (1,100)
30 LET I = I + 1
40 IF I < 7 THEN GOTO 20
```

As you continue to write more and more programs and the length of those programs grows, it will become more and more difficult for you to remember the details of each program.

When the length of your programs are 100 statements or more, it is likely that you will forget exactly what each program does.

A REM or REMARK statement is one in which the programmer makes a note concerning the nature of a program or a certain section of the program. The only time a REM statement is seen is when a LISTing of the program is produced.

When the computer encounters a REM statement, it ignores everything on the line following REM and immediately skips to the next computer line.

Examples of REM statements include:

```
10 REM THIS IS A COMMENT
15 REM A COUNTING PROGRAM
100 REM THIS SECTION BEGINS AN ENDLESS LOOP
```

The length of a REMark or REM statement may be as short as you wish or as long as three lines. If your comment is longer than three lines, you must begin another program line with REM. As long as you begin a program line with REM, the information entered will not affect the program's operation.

Often, beginning programmers will forget to include line numbers. They are so intent on writing the program that they forget the number. The line is executed instead of being added to the program, sometimes with disastrous results. Boy, does this cause some frustration!

This may have happened to you already. Perhaps, you were typing in one of our examples and made a mistake. You corrected the mistake, but forgot to include the line number.

To help avoid this problem and to make programming easier, BASIC XL comes with a statement called NUM. This statement will automatically number each program line. To use NUM, simply type NUM and push [RETURN]. First, you will notice the READY prompt, and where you would normally expect to find the cursor, will be a line number. Next, on the same line, you will see a blank space and the cursor. You may now type in a program statement.

NUM may be used in four different ways. First, NUM may be used alone. This will produce program lines beginning with 10 and will continue in increments of 10.

NUM

Or you may use NUM followed by a single number. When the computer finds NUM and a single number, it will begin the first program line with the specified number. For example:

NUM 3

the computer begins numbering with 3 and adds lines in increments of 10. The first line would be 3 followed by 13, 23, 33, etc.

If NUM is followed by two numbers separated by a comma, the computer will begin numbering line with the first specified number and increase in increments specified by the second number.

NUM 1000,2

begins numbering with line 1000 and increases in increments of 2. The first line would be 1000 followed by 1002, 1004, 1006, etc.

Finally, NUM may be followed by a comma and a single number. In this case, the computer would start numbering program lines with 10 and would increase in increments specified by the number following the comma.

NUM, 4

would begin numbering with 10 and increase in increments of 4. The first line would be 10 followed by 14, 18, 22, 26, etc.

What happens if a program is in the computer's memory and you want to use NUM?

Instruction:

You type:

```
10 GOTO 20: REM LINE10
20 GOSUB 30 :REM LINE 20
NUM
```

What happens to the program lines? By using the command NUM, you force the computer to NUMBER your program lines for you. The computer begins with line 10 and increases in increments of 10. The number 30 and the cursor should now indicate where to begin the next program line.

Instruction:

You type:

```
30 IF TRUE THEN GOTO 10 :REM LINE 30
NUM 15,5
15 []
REM THIS IS THE ORIGINAL LINE 15
```

(Please keep this program in the computer's memory until you complete the section on RENUM.)

When you type in NUM 15,5, the computer displays the number 15 followed by a blank space and the cursor. You then add the REM statement shown above. When you press {RETURN}, what happens? The ready prompt appears and the computer does not continue to number lines. This is due to the fact that a line 20 already exists in the program.

NUM will stop to avoid a collision with any program line already in the computer's memory. You will recall from previous discussions that each line number must be distinct. If a line number is repeated the computer will only store the line most recently entered.

RENUM is closely related to NUM.

Instruction:

You type:

Computer Responds

RENUM
LIST

```
10 GOTO 30
20 REM THIS IS
   THE ORIGINAL
   LINE 15
30 GOSUB 40
40 IF TRUE THEN
   GOTO 10
```

READY
[]

Notice what happens to the line numbers. The original line 10 is still there. However, the GOTO 20 statement has been changed to GOTO 30. Why? Because originally, the GOTO specified line 20, but it (line 20) has been RENUMbered to line 30. Also, what was line 15 has become line 20.

RENUM automatically changes the numbers of all GOTO and GOSUB statements.

When used alone, RENUM begins with line number 10 and increases in increments of 10.

When RENUM is followed by one number, the computer begins with the specified number and continues to number program lines in increments of 10.

RENUM 1

begins with line 1 and increases in increments of 10. In the above example, the first line would be 1,11,21,31,etc.

RENUM, when followed by a comma and a number begins numbering line with 10 and continues to number lines in increments of the specified number.

RENUM ,8

begins with 10 and increases in increments of 8. The first line would be 10 followed by 18, 26, 34, etc.

Finally, RENUM followed by two numbers separated by a comma will begin numbering lines with the first specified number and will continue to number lines in increments of the specified second number.

RENUM 1000, 4

begins with the number 1000 and increases in increments of 4. The first line would be 1000 followed by 1004, 1008, 1012, etc.

Exercises:

1) Use NUM to enter a program. The program doesn't need to make sense, but be sure to include some GOTO or GOSUB statements.

2) Use RENUM on your program. Try various combinations of starting line number and increment. Be sure to LIST your program each time to see the effects of RENUM.

CHAPTER XXVIII

THE JOYSTICK: MANUAL OR AUTOMATIC

Glossary:

HSTICK -----	A function which reads the positions of the joystick in terms of horizontal positions.
VSTICK -----	A function which reads the positions of the joystick in terms of vertical positions.
STRIG -----	A function which determines if the red button or trigger on the joystick is being depressed.

Throughout this book we have attempted to increase your program skills. In our introduction we indicated that a video game would be the final chapter of this book. But as you probably know, a video game often uses a joystick. A joystick is a device which may be attached to your Atari Home Computer. Actually, your home computer is capable of handling four joysticks. The joysticks are attached in the front part of the computer just below the keyboard. Each has its own number starting with 0, 1, 2, and 3. The joystick is often used to play games on your home computer and was first popularized with the Atari 2600.

In our explanations of the various programs throughout this book, we have stressed the need to consider all possibilities. We have asked you to RUN your programs with diverse responses. This was done to insure that you had allowed for all possible answers.

By using the joystick, you remove the keyboard as the means of interaction between the user and the computer. This limits the number of possible extraneous choices. Each potential choice requires that the programmer check for an inappropriate answer. If this is not done, your program will have "bugs" in it. In case you have not heard this term before, a bug is a programming problem. Good programs do not have "bugs".

But remember Richard's Rule:
Every significant program has a
bug in it.

There's also Card's Corollary: A
program without a bug is
insignificant.

VSTICK and HSTICK are functions which allow the programmer to use the joystick. These functions read the joystick positions. The joystick permits movement that is up and down or right and left. In addition, the computer interprets the movement of the joystick as a series of numbers between 5 and 15. This limits the number of user choices.

VSTICK indicates the vertical position of the joystick, while HSTICK indicates the horizontal position. HSTICK(0) reads joystick number zero (which, strangely, is marked joystick 1 on the computer). If the stick is pushed left, the function "reads" a minus one for use by your program. If it is pushed right, your program reads a plus one. If it is in the middle, you get a zero. Try this with a joystick plugged into the first joystick socket:

```
10 PRINT HSTICK(0)
20 GOTO 10
RUN
```

Push the stick left and right and see what the program PRINTs. (Hit [BREAK] to quit.) Of course, HSTICK(1) reads the second joystick, HSTICK(2) reads the third and HSTICK(3) reads the fourth (if your machine has a third and a fourth socket).

VSTICK is similar in its workings. It reads plus one, zero, or minus one if the joystick is pushed up, centered, or pushed down, respectively. Again, try this:

```
10 PRINT VSTICK(0)
20 GOTO 10
RUN
```

Another function used in conjunction with the joystick is STRIG. The STRIG function is used to indicate whether the red button or trigger of the joystick is depressed.

For reasons having to do with the hardware of the Atari, the STRIG function returns a zero when the stick trigger is pushed and a one when it is not. Try this little experiment:

```
10 PRINT STRIG(0)
20 GOTO 10
RUN
```

Push the button on your joystick and see what happens.

Using these functions, it is relatively easy to write programs using the joystick.

Instruction:

You type:

```
NEW
5 REM THIS PROGRAM DRAWS USING THE JOYSTICK
10 GRAPHICS 7
20 COLOR 1
30 LET X=0 : LET Y=0
40 PLOT X,Y
50 LET X= X+HSTICK(0)
60 LET Y=Y-VSTICK(0)
70 IF X < 0 THEN X=0
80 IF X > 159 THEN X=159
90 IF Y < 0 THEN Y=0
100 IF Y > 79 THEN Y=79
110 IF STRIG(0)=0 THEN GOTO 10
120 GOTO 40
```

Let's examine this program:

line 5- is our REM statement to remind us what this program produces.

line 10- is our GRAPHICS statement and permits us to use mode 7.

line 20- is our COLOR statement and permits us to PLOT with light green.

line 30- initializes our variables X and Y at 0.

line 40- is our PLOT statement.

line 50- changes the value of X to be the same as the value of X plus the horizontal reading of the joystick.

line 60- changes the value of Y to be the same as the value of Y plus the vertical reading of the joystick.

line 70- limits the value of X to numbers that are not less than 0.

line 80- limits the value of X to numbers that are not greater than 159.

line 90- limits the value of Y to numbers that are not less than 0.

line 100- limits the value of Y to numbers that are not greater than 79.

lines 70-100 are there to prevent ERROR 141. If the value of X and/or Y go beyond the limits of graphics mode 7 an error will result. The error will prevent the program from continuing.

line 110- if the trigger is pushed, clears the screen and starts the program again.

line 120- forces the computer to return to line 40 and allows for the continuous execution of this program.

Exercises:

1) Modify the above example so color is plotted only when the trigger is pushed. (Remove line 110, which clears the screen if the trigger is pushed, since the two uses are incompatible.)

2) Change the program to simply show where the joystick has "moved" to by "flashing" the spot which would have been plotted (in the original program).

3) This is tricky: add to exercise 2 to allow the user to change colors by pushing the joystick button.

CHAPTER XXIX

A Real Live Video Computer Game: SNAILS' TRAILS

Well, the hard work is all done. If you have read and understood all the previous chapters, you are ready for a break. And, rather than present you with more statements, functions, keywords, etc., we thought that a game would be both instructive and fun. So here, in its entirety, is the BASIC XL version of a perennially popular video game.

Games similar to "SNAILS' TRAILS" have been around for a long time. A version called "SURROUND" was one of the first games available for the Atari 2600 game machine. But, in the tradition of the video game industry, we should present a storyline:

You are a giant, mutant snail. Wherever you travel, you leave a trail of radioactive slime behind. So poisonous and impenetrable is this slime that should any being (including you, yourself) touch it, it dies instantly. (Yes, yes. If it's that poisonous, how could you lay the trail in the first place? How should we know...YOU are the mutant.)

Further, the scientists of far off H'tra-E have discovered your kind and have imprisoned you and another of your race in a large rectangular arena. Unfortunately, both of you are neither male or female. Instead, you are each a St'i, specially bred to do battle until death! You do not know the meaning of the word P'ot's ("stop")!

So, as the scientists release you from stasis (you can hear the three bells as the stasis field is lifted), you begin by charging straight toward your opponent. But wait! A bit of intelligence enters your crazed brain. If your slime trail is so deadly, perhaps you can entice your enemy to run into your trail, thus killing st'i without damage to yourself. Great strategy!

What's this, though? Your opponent has developed the same strategy. Now you and st'i must race around the arena, with the strategic goal of forcing each other to touch a poisonous trail or to run into the electrified outer fence. (Well, we had to keep you in the arena somehow, didn't we?) But tactics can be important here as well. Look, you are running straight across the arena. At the last second, you veer in front of your enemy! He can't avoid your trail in time! He's going to...Oops. You forgot about the wall. Too bad. R.I.P.

To make a long story into a short game, you and another human opponent must each use an Atari joystick (plugged into joystick ports 1 and 2) to control your snail. The first snail to run into a slime trail or a wall loses, and the other snail scores a point. The first snail to score 10 points wins the game. Good luck.

```

10 REM ---SNAILS' TRAILS---
20 REM
30 REM (FROM THE MOVIE OF THE SAME NAME
40 REM
50 REM THIS GAME REQUIRES 2 JOYSTICKS PLUGGED INTO CONNECTORS
60 REM NUMBER 1 AND 2 (STICKS 0 AND 1)
70 REM
80 REM
90 FAST
100 REM ROUND INITIALIZATION
110 GRAPHICS 5
120 REM (DO SETCOLORS HERE IF DESIRED
130 COLOR 1:PLOT 0,0:DRAWTO 0,39:DRAWTO 79,39
140 DRAWTO 79,0:DRAWTO 0,0
150 H0=20:V0=20:H1=20:V1=20
160 HMV0=1:VMV0=0:HMV1=-1:VMV1=0
170 COLOR 2:PLOT H0,V0:COLOR 3:PLOT H1,V1
180 PRINT "  SCORE":PRINT "PLAYER 1",,"PLAYER 2"
190 PRINT SCORE0,,SCORE1
200 REM START A ROUND
210   FOR FREQ = 50 TO 150 STEP 50
220     FOR VOLUME=15 TO 0 STEP -0.1
230       SOUND 0,FREQ,10,VOLUME
240       NEXT VOLUME
250     NEXT FREQ
300 REM MAIN MOVE LOOP
310 FOR MOVE=1 TO 255 STEP 3
320 REM SENSE AND MOVE PLAYER 0
330 IF HSTICK(0) THEN HMV0=HSTICK(0):VMV0=0
340 IF VSTICK(0) THEN VMV0=VSTICK(0):HMV0=0
350 H0=H0+HMV0:V0=V0-VMV0
360 LOCATE H0,V0,BANG0:IF BANG0 THEN 400
370 COLOR 2:PLOT H0,V0
400 REM SENSE AND MOVE PLAYER 1
410 IF HSTICK(1) THEN HMV1=HSTICK(1):VMV1=0
420 IF VSTICK(1) THEN VMV1=VSTICK(1):HMV1=0
430 H1=H1+HMV1:V1=V1-VMV1
440 LOCATE H1,V1,BANG1:IF BANG1 THEN 500
450 COLOR 3:PLOT H1,V1
500 IF BANG0 OR BANG1 THEN 600
510   FOR VOLUME=14 TO 0 STEP -2
520     SOUND 0,MOVE,10,VOLUME
530     NEXT VOLUME
590 NEXT MOVE:GOTO 300
600 REM SOMEBODY GOT BANGED
610 IF BANG0 AND BANG1 THEN 100

```

```

620 IF BANG0 THEN SCORE1=SCORE1+1
630 IF BANG1 THEN SCORE0=SCORE0+1
640   FOR VOLUME=15 TO 0 STEP -.25
650   IF BANG0 THEN SETCOLOR 1,4,VOLUME
660   IF BANG1 THEN SETCOLOR 2,4,VOLUME
670   SOUND 0,23,0,VOLUME
680   NEXT VOLUME
700 REM CHECK FOR END OF GAME
710 IF SCORE0<10 AND SCORE1<10 THEN 100
720 GRAPHICS 2
730 IF SCORE0>SCORE1 THEN 770
740 PRINT #6;"PLAYER 2 WINS:"
750 PRINT #6,SCORE1;" TO ";SCORE0
760 GOTO 800
770 PRINT #6;"PLAYER 1 WINS:"
780 PRINT #6,SCORE0;" TO ";SCORE1
800 REM END OF A GAME
810 PRINT "PUSH JOYSTICK BUTTON TO PLAY AGAIN"
820 REM WAIT FOR A BUTTON TO BE PUSHED
830 IF STRIG(0) AND STRIG(1) THEN 820
840 RUN

```

How SNAILS' TRAILS Works

 You don't have to understand how the program works to enjoy SNAILS' TRAILS. Just type in the listing and RUN it. But, if you believe you are ready to understand the programming techniques involved, read on. We will explain the program a line (or short group of lines) at a time. The numbers in front of each paragraph below indicate the lines which are being explained.

10-80. Just some REMarks, to remind us what the name of the program is and to give some instructions for running the game. These lines are unnecessary, but it is good practice to always have some such similar comments.

90. Actually, we cheated. This is a new statement, not introduced previously. FAST has the unique ability to speed up your program considerably. May we suggest that you leave out this line until the program is working properly. Then simply add this line and RUN the game again. Look at how much faster it plays. We would STRONGLY recommend that you do NOT use FAST in your own programs until after you have read the description of how it works in the BASIC XL REFERENCE MANUAL.

100. This REMark denotes the beginning of the real work of each round (or turn) in the game. (Remember, the first to win 10 rounds wins the game.)

110. We chose an intermediate pixel size for our display. This program will work in GRAPHICS 7 instead, but you will have to adjust the values in lines 130 through 150.

120. The colors chosen by default when the GRAPHICS statement is executed seemed adequate to us. If you would like, though, you could place one or more SETCOLOR statements on this line.

130-140. We draw a line around the outside of the screen to define the "arena".

150. The H'tra-E scientists decided to start the snails at the same positions each time: halfway down the arena and about a quarter of the way in from each side. H0 and V0 designate the starting Horizontal and Vertical position of the first player while H1 and V1 refer to the second player. (We use 0 and 1 to correspond to joysticks 0 and 1, even though we call the players "first" and "second". The English language doesn't like "zeroeth" and "oneth" very well.)

160. Since the fighting instinct is strong when the snails are first released, they start moving directly toward each other. The left hand snail (player) moves to the right, as indicated by HMOV0 (Horizontal Movement of player 0) being set to plus one. The other player moves left (HMOV1 = -1). Neither snail moves vertically, so the two vertical movements are set to zero.

NOTE: Lines 100 through 160 have, for all intents and purposes, defined the limits and scope of our game. From here on out, the game simply follows the rules we have defined.

Often, the hardest part about writing a program is designing the initialization code. Consider, as an example, the task of writing a version of the classic PONG game.

The movement, per se, is easy: two paddles move up and down the left and right edges of the screen and a ball bounces between them. But consider how much the decisions made at set up time affect the game: How big is the ball? How big are the paddles? Where does the ball start from? Where do the paddles start? Are there boundary walls? If so, where?

When you design and write your own programs, try to pay special attention to the "start up" code. Any extra time spent on this phase will pay you dividends in the form of better looking and more readable programs. And, who knows, your program may even work the first time!

170. We place the snails on their starting spots and draw the first pixel of their slime trails. See how easy this was, once the initialization code had set up everything? Note that player 1 will use color 2 while player 2 uses color 3. Any reason? Not really. Change the colors if you like.

180-190. We display the scores in the text window at the bottom of the screen. The commas and spaces used in these lines are not necessary, but they make the display look better.

Note that we depend upon SCORE0 and SCORE1 being zero when the program is RUN. This is a legitimate use of our knowledge of how BASIC XL works.

200-250. Just before the stasis field is lifted, three bells ring. Remember?

210 and 250. This outer loop generates three FREQuencies. Do you see why they are 50, 100, and 150? Do you remember that higher numbers imply lower frequencies when pure tones (sound quality = 10) are used?

220-240. This inner loop simply makes a tone using sound register zero. Notice how the STEP of -0.1 implies that the volume will take on values of 15.0, 14.9, 14.8, etc., in succession, until it reaches zero. When a pure tone is sounded like this, a rapidly decreasing volume produces a ringing, bell-like note.

300-310. The players are moved once each time we go through this loop. For an explanation of why MOVE is STEPped by 3, see the discussion of line 520. Also, see the discussion of line 590 for more information about this loop.

320-370. This section of the code controls the movement of the first player's snail.

330. If the first joystick (number 0) is pushed left or right, we change the horizontal movement (HMOV0) of the snail to correspond. Since we don't allow diagonal movement, we cancel any vertical movement there might have been (we set VMV0 to zero).

340. Similarly, if the joystick is pushed up or down, we change the vertical movement (VMV0) and cancel the horizontal movement (set HMOV0 to zero). Notice that this implies that a stick pushed diagonally will produce only vertical motion. If you would rather let horizontal movement have priority over vertical, simply swap the order of lines 330 and 340.

350. The most important line of the player movement code. The new player position (H0 and V0) is found by adding the old position (also H0 and V0) to the requested movements (HMOV0 and VMV0).

NOTE the minus sign used in determining the vertical position. It is there because VSTICK returns a plus one when the stick is pushed up, while the Atari graphics system insists that larger numbers are further "down" the screen. This simple little trick keeps everything running smoothly and as the player(s) would expect.

360. Remember that the pixel at H0 and V0 is where this player's snail expects to move to next. What if something is already at that spot, such as a wall or a slime trail? Then the player must "die".

The LOCATE statement puts the color of the pixel at H0 and V0 into the variable BANG0. If nothing already exists at the location, the color will be 0 (the background color), and the program will fall through to line 370. If something exists, though, we are done moving this player, so we go directly to line 400.

370. If we get here, there was no conflicting object in our pixel, so we lay down another piece of our slime trail.

400-450. This is the same code as lines 320 through 370 except that all the references are now to the second player (H1, V1, etc.). We will not explain the code line by line since it is virtually identical.

500. If either player hit something, this must be the end of a round. We exit the main code by simply GOING TO line 600.

510-530. If we get here, both players are still alive. We make a little bell tone to indicate that we have moved. Eliminating these lines will speed the game up significantly. But it won't sound as good.

520. The use of MOVE here needs a special comment. In line 310, we began a FOR...NEXT loop involving MOVE:

FOR MOVE = 1 TO 255 STEP 3

But that only allows 85 moves! Surely we expect that some games will last longer than that, don't we? Sure, but we wanted to use the MOVE variable to determine the frequency of the short bell tone, and SOUND only allows frequency values from 0 through 255. By using the loop shown, we are guaranteed legal sound values.

590. And this, then, is the end of the MOVE loop. But what happens if both players are still alive at the end of 85 turns? Simple answer: GO back TO line 300 and start the FOR...NEXT loop over again. Actually, if we didn't need sounds, this line could consist of just the GOTO and we could eliminate the FOR...NEXT entirely.

600-680. When one or both of the players runs into something, we execute this code.

610. If both players hit at the same time, don't do anything. Just go remove the stasis field again.

620-630. Add a point to the score of the player that did NOT hit anything.

640-680. Another decreasing volume loop. This time, the sound quality (line 670) is set to zero so we get noise. The effect is something like an explosion. 650-660. Depending on which player hit something, we flash one or the other of the slime trails red. Why do we use "VOLUME" for what is normally luminance? Simply because it happened to be changing and in the range required for luminance. The result: a bright flash which quickly fades out. Nice.

700-710. If neither player has yet scored 10 points, we start another round at line 100.

720. If somebody lost, we might as well broadcast the fact in great big letters. Remember GRAPHICS 2 produces large characters in the graphics window and retains the text window.

730. What message is displayed depends on who won.

740-750. The message, in nice big characters. Notice the semicolon in 740 forces the first line to the left edge of the screen. The comma in 750 places the scores in the middle. The result is an acceptable score display.

760. We bypass the other message by GOing TO the end-of-game code.

770-780. The same code as lines 740 and 750, changing who won.

800-810. Just in case, we allow the H'tra-Ethlings to start with more fresh snails. Note that the message in line 810 is printed at the bottom of the screen, in the text window.

820-830. This is actually a very small program loop which will last forever. Unless, of course, one or the other player pushes his/her joystick button.

840. Yes, you really can use RUN as a program statement. In situations such as this, it has the advantage of forcing a clear of all variables, etc.

Of course, if we had wanted to do something like keep track of high scores (inapplicable in this particular game), we would have to use GOTO 10 here. The further implication is that we could not assume any variables to be zero when a game started. (See the comments about line 190.)

Some Last Comments

Are you really still with us? If you got lost in that explanation, perhaps the best thing to do would be to type in the game and try it. May we strongly suggest that you SAVE or CSAVE the program before RUNNING it. If you typed it in exactly as shown, no problems should occur. But...

This kind of game is what programming is all about. Not that everyone should write games. Rather that everyone who is serious about programming should seek a well defined programming goal, design the means to accomplish it, and start coding. And whether a program plays a game or calculates the residual value of fully depreciated Edsels, if it does what it was designed to do, it is a success.

Remember this, then, when you tackle exercise number 3, below.

EXERCISES

1. Change line 120 so that the program will draw a red border around the arena.

2. As this game is written, a player may wipe himself/herself out too easily by backtracking on his/her own slime trail. Obviously, nobody intends to do this, but sometimes when you turn fast you will end up going diagonally and the program's logic will force you back into yourself.

Try to "fix" the program so that a player cannot backtrack so easily. Fair warning: the answer which we provide for this exercise still makes it possible--though much harder--to backtrack. If you do it better, great!

3. Think of a problem which a computer can solve. Write a program which will solve the problem.

The "problem" might be a game you would like to see. Or it might be a loan amortization calculation. Or perhaps a checkbook balancer. Don't limit your horizons.

If you really can't think of a problem you want solved, try some of our suggestions:

Games: TIC-TAC-TOE
 PONG
 CRAPS

(A very easy game to write without graphics. So why not put in some really good pictures?)

Business: LOAN AMORTIZATION
 RATE OF RETURN ON INVESTMENT
 INVOICE PRINTING

(Filing invoices is MUCH more difficult. This book has NOT given you sufficient background to write such a program.)

Home use: CHECKBOOK BALANCER
 RECIPE MULTIPLIER

(What is 6 times $\frac{1}{3}$ cup--in quarts?)

Education: ADDITION DRILL AND PRACTICE

(Also very easy without graphics. But how about showing 3 red apples plus 4 orange oranges?)

CHAPTER XXX

Congratulations! You've made it to the final chapter of our book. Good for you! It took patience, and an open mind and hard work to get this far. Now what? What have you learned? What good does it do you?

In the process of working through this book, you have learned to operate your ATARI HOME COMPUTER. That and a dollar will buy you a cup of coffee. Right? Actually, there is more to it than that.

By learning to operate your ATARI HOME COMPUTER, you need never fear computers again. You now know that a computer will not bite, and in fact, the computer is always READY for your next command.

You have also opened the door to many other possibilities. Your familiarity with the computer and simple BASIC statements will allow you to use commercially produced software with a minimal amount of effort. With each new day more software programs come on the market. You can increase your use of your home computer and at the same time learn a new foreign language like French or Italian. Or you can choose from word processing, home management, home phone lists, and address label programs. Actually, the possibilities are almost endless.

At social gatherings, you can be part of the group discussing programming. Or, you can show your brilliance by word dropping. For example: you could tell your friends about favorite variables; or how IF...THEN and GOSUB...RETURN statements have changed your life. Why, you'll be the center of attention.

Maybe what you have learned is that you aren't meant to be a programmer at all. Knowing that isn't a waste of time. Now that you know, you again can rely on commercially produced software for all your needs. Believe it or not, not everyone was intended to be a programmer. However, everyone should know enough to be computer literate. Assuming you have read and understood most of this book, you are now computer literate.

Finally, if all this programming stuff really got to you, if night after night you spent hours with our book and your ATARI, you may have a future in programming. There is more BASIC to learn and other computer languages.

One final note: if you want to pursue a career in programming or if you want to become a better programmer, there are two things you should do.

First, start reading. Begin by reading your BASIC XL Reference Manual. This will give you information about computer functions and commands not covered in this book. There are many other books on the market that will provide additional information on BASIC. If you don't have an excellent mathematics background, carefully check the books you use. Be sure each book deals with functions and commands and not just higher levels of math. Don't buy a book sight unseen.

Second, start programming. Don't worry about what you programs you write or whether they are useful or not. Any programming experience will be valuable. Also, there are many magazines and user groups that publish programs for you to try. The better publications also include explanations of their programs. Programming is like tennis or golf or music or any other learned skill, the more you do it, the better you'll get at it.

ANSWERS

Answers: Chapter I

1. PRINT "JEFFREY R. SMITH"
2. PRINT "GIVE ME LIBERTY OR GIVE ME DEATH"

Please note: the above answers are correct. However, your screen will look like this:

```
PRINT "JEFFREY R. SMITH"
JEFFREY R. SMITH
```

```
READY
PRINT "GIVE ME LIBERTY OR GIVE ME DEATH"
GIVE ME LIBERTY OR GIVE ME DEATH
```

Answers: Chapter II

1. PRINT "JOHN Q. PUBLIC"
PRINT "924 MAIN STREET, ANYWHERE,USA"
PRINT "800-555-1212"
2. PRINT 15+25
PRINT 38-14
PRINT 680*12
PRINT 25/5
3. PRINT "000-11-2222"

Answers: Chapter III

- 1) a) Assigns the value of 15 to X.
b) Assigns the value of 144 to B.
c) Assigns the value of 338 to F2.
d) Assigns the value of 323 to Z.
(which is 338-15)
- 2) LET TOTAL = 7 + 5
PRINT TOTAL + 1

(N.B.: You could have coded PRINT 7 + 5 + 1, but the answer shown keeps better to the spirit of this chapter.)

3) 198

Answers: Chapter IV

```
1)  10 LET X = 2
    20 LET Y = 4
    30 PRINT X + Y
    40 PRINT X * Y
```

When you typed RUN, did your program work? Good. If not compare your answer to the one given. It is a good idea to get in the habit of testing your programs. Really, that is the only way to know if they work properly.

Since variable names, the value of a variable and line numbers chosen in a program are arbitrary, your program is probably correct if it is similar to the one above, and if it RUNs properly.

2) Assuming the same program is used as was used in the answer to exercise 1:

```
10 LET X = 2
30 PRINT X+Y
40 PRINT X*Y
```

OR

```
10 LET X = 2
20 PRINT X+Y
30 PRINT X*Y
```

Did you correctly note that the computer assumes Y has a value of zero? In Basic XL, all variables have a value of zero until something else is assigned to them.

3) You should have typed in a single line, such as:
20 LET Y = 4 OR 15 LET Y = 4

The important point here is that you used a line number somewhere between the line numbers for the first and third lines. When you RUN the program, you should get the same results as you did in exercise 1.

Answers: Chapter V

You Type:

Computer Responds

- | | | | |
|-------------------------|-----|---|----|
| 1) NEW | | | |
| 10 LET X = 12 | | | |
| 20 LET Y = 8 | | | |
| 30 PRINT X ; Y | | | |
| RUN | 128 | | |
| 2) 30 PRINT X,Y | | | |
| RUN | 12 | 8 | |
| 3) 30 PRINT X+Y,X-Y,X*Y | | | |
| RUN | 20 | 4 | 96 |

PLEASE NOTE: Since variables names, the value of a variable and line numbers chosen in a program are arbitrary, your programs are probably correct if they look similar to the answers shown. Be sure to check that your programs RUN properly.

ANSWERS: Chapter VI

- ```
1) 10 PRINT "N," "N-SQUARED," "N-DOUBLED"
 20 LET N = 1
 30 PRINT N, N*N, N+N
 40 LET N = N + 1
 50 LET N = X
 60 GOTO 30

2) 10 LET X = 1
 20 PRINT X
 30 LET X = X + 1
 40 GOTO 20

3) 10 LET X = 1
 20 PRINT "YOUR NAME",X
 OR
 20 PRINT "YOUR NAME";X
 30 LET X = X + 1
 40 GOTO 20

4) 10 LET X = 1
 20 PRINT "YOUR NAME",X
 30 LET X = X + 2
 40 GOTO 20
```

```

5) 10 LET X = 5
 20 PRINT "YOUR NAME", X
 30 LET X = X + 2
 40 GOTO 20

```

RUN each of your programs.

If your programs look similar to the ones above and if they RUN properly, they are probably correct.

#### Answers: Chapter VII

-----

- 1) 10 LET C = 5  
20 PRINT C  
30 LET C = C + 5  
40 IF C <= 100 THEN GOTO 20
- 2) 10 LET T = 1  
20 PRINT "A COUNTING COMPUTER IS EASY"  
30 LET T = T + 1  
40 IF T <= 10 THEN GOTO 20
- 3) 10 LET K = 937  
20 LET M = 486  
30 IF K = M THEN PRINT "THEY ARE EQUAL"  
40 IF M > K THEN PRINT M, " IS LARGER"  
50 IF K > M THEN PRINT K, " IS LARGER"

If your programs look similar to the ones above and if they RUN properly, they are probably correct.

#### Answers: Chapter VIII

-----

- 1) 10 PRINT "WHAT IS THE BASE"  
20 INPUT BASE  
30 PRINT "WHAT IS THE HEIGHT"  
40 INPUT HEIGHT  
50 PRINT "AREA = ";BASE\*HEIGHT/2
- 2) 10 PRINT "ENTER ONE TEST SCORE"  
20 INPUT TEST1  
30 PRINT "ENTER ANOTHER TEST SCORE"  
40 INPUT TEST2  
50 PRINT "ENTER ANOTHER TEST SCORE"  
60 INPUT TEST3  
70 LET TOTAL = TEST1+TEST2+TEST3  
80 LET AVERAGE=TOTAL/3  
90 PRINT "THE AVERAGE OF THE THREE TEST SCORES IS "; AVERAGE



```

3) 10 PRINT "WHAT IS THE MAXIMUM NUMBER"
 20 INPUT MAX
 30 LET NUM = 1
 40 PRINT NUM, NUM*NUM
 50 LET NUM = NUM + 1
 60 IF NUM <= MAX THEN GOTO 40

```

Answers: Chapter IX

-----

```

1) 10 PRINT "ENTER YOUR BOWLING SCORES"
 20 INPUT YOU1, YOU2
 30 PRINT "ENTER YOUR OPPONENT'S BOWLING SCORES"
 40 INPUT OPP1, OPP2
 50 IF YOU1>OPP1 AND YOU2>OPP2 THEN PRINT "YOU WON
 BOTH GAMES"
 60 IF YOU1<OPP1 AND YOU2<OPP2 THEN PRINT "YOU LOST
 BOTH GAMES"

2) 10 PRINT "PICK A NUMBER"
 20 INPUT X
 30 IF X>10 AND X<20 THEN PRINT "YOUR NUMBER IS
 BETWEEN 10 AND 20"

3) 10 PRINT "GIVE ME 3 NUMBERS, IN ORDER"
 20 INPUT NUM1
 30 INPUT NUM2
 40 INPUT NUM3
 50 IF NUM1 < NUM2 AND NUM2 < NUM3 THEN PRINT
 NUM1,NUM2,NUM3

```

Answers: Chapter X

-----

```

1) 10 LET D = 1
 20 PRINT RANDOM(1,50)
 30 LET D = D + 1
 40 IF D <= 10 GOTO 20

2) 10 LET D = 1
 20 PRINT RANDOM(1000)
 30 LET D = D + 1
 40 IF D <= 10 GOTO 20

3) 10 LET NUMBER = RANDOM(1,50)
 20 PRINT "PICK A NUMBER BETWEEN 1 AND 50";
 30 INPUT GUESS
 40 PRINT GUESS, NUMBER

```

```

4) 10 LET NUMBER = RANDOM(1,50)
 20 PRINT "PICK A NUMBER BETWEEN 1 AND 50";
 30 INPUT GUESS
 40 IF GUESS<NUMBER THEN PRINT "TOO LOW"
 50 IF GUESS>NUMBER THEN PRINT "TOO HIGH"
 60 IF GUESS<>NUMBER THEN GOTO 20
 70 PRINT "YOU GOT IT"

```

```

| Note: If you solved exercise 4 |
| correctly, congratulations are in |
| order. This is the classic "computer |
| number guessing game" and is often the |
| target or end of introductory |
programming classes.

```

Answer: Chapter XI

-----

1) If you are able to copy a program and to retrieve it, you probably understand the use of the program recorder.

Answers: Chapter XII

-----

- 1) If not, try again.
- 2) ERROR -NOT SAVED FILE
- 3) ERROR (unpredictable, but often 137)

Answers: Chapter XIII

-----

1) Obviously, the contents of your letter will differ, but here's an example of proper form. Note how we used commas successfully but did NOT use any at the ends of lines.

```

10 LPRINT "DEAR JOHN,"
20 LPRINT "HOW ARE YOU?"
30 LPRINT "I'M SORRY I HAVE WRITTEN SOONER."
40 LPRINT "RECENTLY, I HAVE BEEN SPENDING MY SPARE
 TIME LEARNING TO PROGRAM MY ATARI."
50 LPRINT "I WILL WRITE YOU A LONG LETTER SOON."
60 LPRINT,, "YOUR FRIEND,"
70 LPRINT,, "JIM"

```

2) By using LIST"P:", the program you typed in should appear on the printer.

--170--

#### Answers: Chapter XIV

-----

- 1) 10 GRAPHICS 7  
20 PLOT RANDOM (0,159), RANDOM (0, 79)  
30 FOR I = 1 TO 25  
40 COLOR RANDOM (1,3)  
50 DRAWTO RANDOM (0, 159), RANDOM (0, 79)  
60 NEXT I  
70 GOTO 10
- 2) 10 GRAPHICS 7  
20 FOR I = 1 TO 15  
30 COLOR RANDOM (1,3)  
40 COL1= RANDOM (160): COL2 = RANDOM (160)  
50 ROW1 = RANDOM (80): ROW 2 = RANDOM (80)  
60 PLOT COL1, ROW1 : DRAWTO COL1, ROW2  
70 DRAWTO COL2, ROW2: DRAWTO COL2, ROW1  
80 DRAWTO COL1, ROW1  
90 NEXT I  
100 GOTO 10
- 3) 10 GRAPHICS 7 : PLOT 0,0  
20 FOR X = 0 TO 39 STEP 2  
30 COLOR RANDOM (1,3)  
40 DRAWTO 159-X,X  
50 DRAWTO 159-X,79-X  
60 DRAWTO X,79-X: DRAWTO X,X+2  
70 NEXT X  
80 GOTO 10

-----  
NOTE: By now, we are presenting  
exercises so complex that many,  
many different "answers" are pos-  
sible. We would suggest that you  
type in our programs only if you  
cannot solve the exercises yourself  
or if you simply want to see what  
kind of result we expected. The  
rule here is simple: If your pro-  
gram works, it is probably right.  
-----

#### Answers: Chapter XVI

-----

- 1) 10 PRINT "ENTER TOTAL WEEKLY SALES"  
20 INPUT SALES  
30 IF SALES >=20000 THEN LET PAY=2000  
40 IF SALES <20000 THEN LET PAY=1000  
50 PRINT PAY

- 2) Add the following lines:  
30 IF SALES>=20000 THEN LET PAY=2000  
50 LET FEDTAX=105 : 0.067\*PAY  
60 LET NETPAY=PAY-FEDTAX-SOCSECRTY  
70 PRINT NETPAY

ANSWER: Chapter XVII

- 1) 150 PRINT "ONE PINT EQUALS 1) 8 OZS. 2) 16 OZS. 3) 24 OZS"  
160 LET CORRECTANSR = 2  
170 GOSUB 400  
180 PRINT "ONE GALLON = 1) 4 QUARTS 2) 4 CUPS 3) 4 PINTS"  
190 LET CORRECTANSR = 1  
200 GOSUB 400

If your answers are similar in format to the ones above  
and if your program runs properly then your program is  
probably correct.

Answers: Chapter XVIII

- 1) 10 FOR I=1 TO 10  
20 PRINT "JOHN SMITH"  
30 NEXT I  
2) 10 FOR X= 10 TO 1 STEP - 1  
20 PRINT X  
30 NEXT X  
40 PRINT "BLAST OFF"  
3) 10 FOR G = 0 TO 100 STEP 5  
20 PRINT G  
30 NEXT G  
4) 10 PRINT "HOW FAR SHOULD I GO";  
20 INPUT MAX  
30 FOR N=1 TO MAX  
40 PRINT N, N\*N  
50 NEXT N

Answers: Chapter XIX

- 1) 10 FOR I=1 TO 9  
20 FOR J=1 TO 9  
30 PRINT I\*J;  
40 PRINT " ",  
50 NEXT J  
60 PRINT  
70 PRINT  
80 NEXT I

Answers: Chapter XX

- ```
-----
1)  10 INPUT INVITEES$
    20 PRINT "DEAR "; INVITEE
    30 PRINT "ON JUNE 15TH I WILL BE 39 ONE MORE TIME".
    40 PRINT "PLEASE ";
    50 PRINT INVITEES$;
    60 PRINT "HELP ME CELEBRATE THIS OCCASION".
    70 PRINT "MY PARTY WILL BE HELD AT THE GREEN OAKS
        COUNTRY CLUB".
    80 PRINT "BE PROMPT";
    90 PRINT INVITEES$;
   100 PRINT "THE PARTY STARTS AT 8:00"
   110 PRINT "I'LL NEED HELP BLOWING OUT THE CANDLES".
   120 PRINT
   130 PRINT "SINCERELY,"
   140 PRINT "JACK BENNY"
   150 PRINT
   160 GOTO 20

2)  10 DIM ANSWERS$(1)
    20 PRINT "DO YOU LIKE ME";
    30 INPUT ANSWERS$
    40 IF ANSWERS$="Y" THEN PRINT "I LIKE YOU, TOO"
    50 IF ANSWERS$="N" THEN PRINT "BRRRRAP!"
```

Answer: Chapter XXI

- ```

1) 10 DIM NAMES$(35)
 20 PRINT "ENTER YOUR NAME"
 30 INPUT NAMES$
 40 LET A= LEN(NAMES$)
 50 FOR I=A TO 1 STEP -1
 60 PRINT NAMES$(I,I);
 70 NEXT I
```

ANSWERS: Chapter XXII

- ```
-----
1)  10 SOUND RANDOM(4), RANDOM(256), RANDOM(16),
    RANDOM(16)
    20 FOR I=1 TO 200: NEXT I
    30 GOTO 10

2)  10 P=RANDOM (256)
    20 SOUND 0,P,10,8
    30 FOR I=1 TO 10 : NEXT I
    40 GOTO 10
```

Answers: Chapter XXIII

```
-----
1) 10 GRAPHICS 1
    20 PRINT #6 "STEVEN LEWIS"
    30 PRINT #6 "YOU ARE A STAR"

2) 10 GRAPHICS 2
    20 PRINT #6;"      THE PALACE      "
    30 POSITION 0,2
    40 PRINT #6;"*****"
    50 PRINT #6;"*      jonathon      *"
    60 PRINT #6;"*      michael's     *"
    90 PRINT #6;"*                      *"
    100 PRINT #6;"*                      *"
    110 PRINT #6;"*  5 performances  *"
    120 PRINT #6;"*      a day        *"
    130 PRINT #6;"*      LIVE         *"
    140 PRINT #6;"*****"
```

Answers: Chapter XXIV

```
-----
1) 10 GRAPHICS 3

    10 GRAPHICS 5
```

The above program line should replace the first line in each of the three exercises listed at the end of Chapter XIV. First, RUN the program in mode 7 and then change to mode 3 and then to mode 5.

```
2) 100 COLOR 3-SCREENCOLOR
    110 PLOT COLUMN,ROW
```

Answers: Chapter XXV

```
-----
1) 10 SETCOLOR 1, RANDOM(16), RANDOM(16)
    20 FOR WAIT=1 TO 1000: NEXT WAIT
    30 GOTO 10

2) 10 GRAPHICS 3
    20 SETCOLOR 4, RANDOM(16), RANDOM(16)
    30 FOR WAIT=1 TO 1000: NEXT WAIT
    40 GOTO 20
```

Line 20 in answer 1 and line 30 in answer 2 are arbitrary delays and do not affect the correctness of the answer. Without some such delay, though, the display color will flicker too fast to be discernable.

Answer: Chapter XXVI

```
-----
1)  10 GRAPHICS 8
    20 COLOR 1
    30 PLOT 0,0
    40 DRAWTO 159,0
    50 DRAWTO 159,79
    60 DRAWTO 0,79
    70 DRAWTO 0,0
```

Answers: Chapter XXVIII

```
-----
1)  40 IF STRIG(0)=0 THEN PLOT X,Y
    110 {RETURN} (deletes line 110)

2)  40 COLOR 1: PLOT X,Y
    46 FOR I=1 TO 200: NEXT I
        (The above line is just a delay loop and is
         not strictly necessary.)
    48 COLOR 0: PLOT X,Y

3)  20 C=1
    40 IF STRIG(0)=0 THEN C=C+1
    42 IF C>3 THEN C=0
    44 COLOR C: PLOT X,Y
```

ANSWERS: Chapter XXIX

```
-----
1)  Change line 120 to read:
    120 SETCOLOR 0,4,4
The hue and luminance numbers ("4,4") might need to be
changed slightly for different television sets or
monitors.

Did you remember that SETCOLOR 0 applied to COLOR 1,
which is used to draw the border?

2)  Change the following lines as shown:
    330 IF HSTICK(0)<>0 AND HMOV0=0 THEN
        HMOV0=HSTICK(0) : VMV0=0
    340 IF VSTICK(0)<>0 AND VMV0=0 THEN
        VMV0=VSTICK(0) : HMOV0=0
    410 IF HSTICK(1)<>0 AND HMOV1=0 THEN
        HMOV1=HSTICK(1) : VMV1=0
    420 IF VSTICK(1)<>0 AND VMV1=0 THEN
        VMV1=VSTICK(1) : HMOV1=0
```

Do you see why those changes work? Basically, we can read line 330 to say "if the first user is pushing the stick left or right AND if he was NOT moving left or right before now, then change his movement to be left or right (according to how the stick is pushed) and cancel his vertical movement." The other lines may be read in a similar fashion.

How does this deceptively simple change work? Well, if the user's snail was going right (for example), then pushing the joystick left OR right has no effect. Reasons: if he is already going right, why do anything; if he is going left, we don't want him to run into himself (the conditions of this exercise), so we disallow the movement.

The flaw in this logic is complex. If the snail is moving up and the user pushes the joystick diagonally down and to the left (for example), our program will see and allow the left movement first (HMOV0 will be set to -1 and VMV0 will be set to 0 in line 330). But then when the program reaches the next line, it will also see and allow the down movement (incidentally cancelling the left movement), and our snail will seem to double back on himself. Note, however, that as this program is written the flaw does not exist for horizontal movement.

Can the flaw be fixed? Yes, at the expense of more complex logic, resulting in a program which runs slower. Is it worth fixing? Probably not, since the game is very playable as is.

A final comment. There were several possible answers to this exercise, so if your version worked it is probably correct. We present here several possible choices for line 330 (the other lines would be similar).

```
330 IF HMOV0=0 AND HSTICK(0)<>0 THEN
      HMOV0=HSTICK(0) : VMV0=0
330 IF HSTICK(0) AND NOT HMOV0 THEN
      HMOV0=HSTICK(0) : VMV0=0
330 IF HSTICK(0) THEN IF NOT HMOV0 THEN
      HMOV0=HSTICK(0) : VMV0=0
330 IF NOT HMOV0 THEN IF HSTICK(0) THEN
      HMOV0=HSTICK(0) : VMV0=0
```

Incidentally, the last two examples will probably run a little faster than the others, since the second function will not even be checked unless the first expression is true, thanks to the extra THEN we used.

A REFERENCE MANUAL

for

BASIC XL

This book is Copyright (c) 1983 by
Optimized Systems Software, Inc.
1221-B Kentwood Avenue.
San Jose, CA 95129

Portions of this book are
Copyright (c) 1980 Atari, Inc.
and are reprinted with the
permission of Atari, Inc.

All rights reserved. Reproduction or
translation of any part of this work beyond
that permitted by sections 107 and 108 of the
United States Copyright Act without the
permission of the copyright owner is
unlawful.

ACKNOWLEDGEMENT

OSS gratefully acknowledges the cooperation of Atari, Incorporated, for the kind permission to reprint portions of the Atari BASIC Reference Manual. Please be aware that these portions have been copyrighted by Atari, Incorporated, and respect the rights implied thereby.

CAVEAT

Every effort has been made to ensure that this manual accurately documents the language BASIC XL. However, due to the ongoing improvement and update of all OSS, Inc., software, we cannot guarantee the accuracy of printed material. OSS, Inc., disclaims all liability for changes, errors, or omissions, either in the documentation or in the software product itself.

TRADEMARKS

BASIC XL, MAC/65, DOS XL, OSS, and SuperCartridge are trademarks of Optimized Systems Software, Inc.

Atari is a registered trademark of Atari, Inc.

The following are trademarks of Atari, Inc.:

Atari 400 Home Computer	Atari 810 Disk Drive
Atari 800 Home Computer	Atari 850 Interface Module
Atari 1200XL Home Computer	Atari 1050 Disk Drive
Atari 400	Atari 810
Atari 1200XL	Atari 850
	Atari 1050

TABLE OF CONTENTS

Chapter 1	Introduction	1
1.1	Features of BASIC XL	1
1.2	Special Notations	2
1.3	Glossary and Terminology	3
1.4	Operating Modes	7
Chapter 2	Variables, Operators, Expressions	9
2.1	Variables (var)	9
2.1.1	Arithmetic Variables (avar)	10
2.1.2	Arrays and Matrices (mvar)	10
2.1.3	String Variables (svar)	12
2.1.4	String Array Variables (svar)	12
2.1.5	DIM	13
2.2	Operators	14
2.2.1	Arithmetic Operators (aop)	14
2.2.2	Logical Operators (lop)	15
2.2.3	Operator Precedence	16
2.3	Expressions (exp)	17
2.3.1	Numbers	17
2.3.2	Arithmetic Expressions (aexp)	18
2.3.3	String Expressions (sexp)	19
Chapter 3	Program Development Commands	21
3.1	BYE	21
3.2	CLR	21
3.3	CONT	22
3.4	DEL	22
3.5	DOS / CP	23
3.6	FAST	23
3.7	LIST	24
3.8	LOMEM	24
3.9	LVAR	25
3.10	NEW	25
3.11	NUM	25
3.12	REM	26
3.13	RENUM	27
3.14	RUN	27
3.15	SET	28
3.16	STOP	31
3.17	TRACE / TRACEOFF	31
Chapter 4	Program Control Statements	33
4.1	Assignment Statement	33
4.2	END	34
4.3	FOR...TO...STEP / NEXT	35
4.4	GOSUB / RETURN	36
4.5	GOTO	37
4.6	IF...THEN	39
4.7	IF...ELSE...ENDIF	40
4.8	LET	41
4.9	MOVE	42
4.10	ON...	43
4.11	POP	44
4.12	RESTORE	45
4.13	TRAP	45
4.14	WHILE / ENDWHILE	46

Chapter 5	Input/Output Commands and Devices	47
5.1	Comments and Notations	47
5.2	BGET	49
5.3	BPUT	50
5.4	CLOAD	50
5.5	CLOSE	50
5.6	CSAVE	51
5.7	DATA	51
5.8	DIR	52
5.9	ENTER	52
5.10	ERASE	53
5.11	GET	53
5.12	INPUT	53
5.12.1	Advanced use of INPUT	54
5.13	LOAD	55
5.14	LPRINT	55
5.15	NOTE	55
5.16	OPEN	56
5.17	POINT	57
5.18	PRINT	57
5.19	PRINT USING	58
5.20	PROTECT	63
5.21	PUT	63
5.22	READ	63
5.23	RENAME	64
5.24	RGET	64
5.25	RPUT	65
5.26	SAVE	66
5.27	STATUS	66
5.28	TAB	66
5.29	UNPROTECT	67
5.30	XIO	67
5.31	An Example Program	68
Chapter 6	Function Library	69
6.1	Arithmetic Functions	69
6.1.1	ABS	69
6.1.2	CLOG	69
6.1.3	EXP	70
6.1.4	INT	70
6.1.5	LOG	70
6.1.6	RANDOM	70
6.1.7	RND	71
6.1.8	SGN	71
6.1.9	SQR	71
6.1.10	An Example Program	71
6.2	Trigonometric Functions	72
6.2.1	ATN	72
6.2.2	COS	72
6.2.3	DEG / RAD	72
6.2.4	SIN	72
6.2.5	An Example Program	73

6.3	String Functions	73
6.3.1	ASC	73
6.3.2	CHR\$	73
6.3.3	FIND	74
6.3.4	LEFT\$	75
6.3.5	LEN	75
6.3.6	MID\$	75
6.3.7	RIGHT\$	76
6.3.8	STR\$	76
6.3.9	VAL	76
6.3.10	An Example Program	77
6.4	Game Controller Functions	78
6.4.1	HSTICK	78
6.4.2	PADDLE	78
6.4.3	PEN	78
6.4.4	PTRIG	78
6.4.5	STICK	79
6.4.6	STRIG	79
6.4.7	VSTICK	79
6.4.8	An Example Program	80
6.5	Player/Missile Functions	80
6.5.1	BUMP	80
6.5.2	PMADR	81
6.6	Special Purpose Functions	81
6.6.1	ADR	81
6.6.2	DPEEK	81
6.6.3	DPOKE	82
6.6.4	ERR	82
6.6.5	FRE	82
6.6.6	HEX\$	83
6.6.7	PEEK	83
6.6.8	POKE	83
6.6.9	SYS	84
6.6.10	TAB	84
6.6.11	USR	84
6.6.12	An Example Program	86
Chapter 7	Screen Graphics and Sound	87
7.1	GRAPHICS	87
7.1.1	GRAPHICS Mode 0	88
7.1.2	GRAPHICS Modes 1 and 2	88
7.1.3	GRAPHICS Modes 3, 5, and 7	89
7.1.4	GRAPHICS Modes 4 and 6	89
7.1.5	GRAPHICS Mode 8	90
7.1.6	GRAPHICS Modes 9, 10, and 11	90
7.2	COLOR	91
7.3	DRAWTO	92
7.4	LOCATE	92
7.5	PLOT	93
7.6	POSITION	93
7.7	PUT/GET as Applied to Graphics	93
7.8	SETCOLOR	94
7.9	XIO Special Fill Application	96
7.10	SOUND	97

Chapter 8	Player / Missile Graphics	99
8.1	Overview of P/M Graphics	99
8.2	P/M Graphics Conventions	101
8.3	BGET and BPUT with P/M's	101
8.4	PMCLR	102
8.5	PMCOLOR	102
8.6	PMGRAPHICS	102
8.7	PMMOVE	104
8.8	PMWIDTH	105
8.9	POKE and PEEK with P/M's	105
8.10	MISSILE	105
8.11	MOVE with P/M's	106
8.12	USR with P/M's	106
8.13	Example PMG Programs	107
Appendix A	Error Descriptions	111
Appendix B	System Memory Locations	116
Appendix C	BASIC XL Memory Map	119
Appendix D	ATASCII Character Set	121
Appendix E	Syntax Summary and Keyword Index	127
Appendix F	Compatibility with Atari BASIC	131

1.1 Features of BASIC XL

Compatibility with Atari BASIC

Because BASIC XL uses the same tokens as Atari BASIC, programs written in Atari BASIC which have been SAVED can be LOADED and RUN using BASIC XL.

FAST Program Execution

BASIC XL allows you to RUN your programs faster than ever with the new FAST command, thus making games written in BASIC almost as fast as arcade games.

Easy Program Formatting

Unlike other BASICs, BASIC XL does not care whether you use upper or lower case letters when you enter your programs. This alone makes programs more readable. However, BASIC XL does even more. It will automatically prompt you with line numbers or renumber an entire program at your request. Also, the LIST command has a program formatter built in, so your programs are easier to follow, no matter how complex or involved they are.

Built-in Functions

BASIC XL contains over 40 built-in functions covering a wide range of applications. The chapter titled FUNCTION LIBRARY explains these functions and their usages.

Graphics

BASIC XL offers the same bit-map graphics manipulation available in Atari BASIC, and allows amazing flexibility in color choice and pattern variety. Chapter 7 explains each command and gives examples of the many ways to use each.

Player / Missile Graphics

BASIC XL allows you easy access to the player / missile graphics available on the Atari through the use of built-in functions and commands. With BASIC XL, p/m graphics are as easy to control as common bit-map graphics.

Game Controllers

Not only does BASIC XL support the game controller functions as Atari BASIC, but it also adds some other game controller functions which make interpreting and using the joysticks much easier.

Sound

The Atari Personal Computer is capable of emitting a large variety of sounds including simulated explosions, electronic music, and "raspberries", and BASIC XL allows you to have control over these sounds available.

Wraparound and Keyboard Repeat

If you enter a program line which is longer than the length of the screen, the line "wraps around" to the next line so that you can view it. Also, if you hold down any key for over 1/2 second, it will start repeating.

Error Messages

If a data entry error is made, the screen display shows an error message and the line on which the error occurred (with the character at which the error occurred highlighted). Most errors will also display a short, descriptive message along with the error number. Appendix A contains a list of all the error messages and their explanations.

1.2 Special Notations used in this Manual

Line Format

The format of a line in a BASIC program includes a line number (abbreviated to lineno) at the beginning of the line, followed by a statement keyword, followed by the body of the statement and ending with a line terminator command (<RETURN> key). In an actual program, the four elements might look like this:

	Statement	Statement	
lineno	Keyword	Body	Terminator
-----	-----	----	-----
100	PRINT	A/X*(Z+4.567)	<RETURN>

Several statements can be typed on the same line provided they are separated by a colon (:).

Capital Letters

In this book, all keywords and functions are printed in uppercase to differentiate them from the other parts of a statement.

Lower Case Letters

In this manual, lower case letters are used to denote the various classes of items which may be used in a program, such as variables (var), expressions (exp), and the like.

Items in Brackets

Brackets ([]) contain optional items which may be used, but are not required, in the format of a statement. If the item enclosed in brackets is followed by three dots (e.g. [exp,...]), more than one of that item may be entered, but none are required.

Items Stacked Vertically in Bars

Items stacked vertically in bars indicate that any one of the stacked items may be used, but that only one at a time is permissible. In the example below, type either the GOTO or the GOSUB.

```
100   | GOTO   | 2000  
      | GOSUB  |
```

Command abbreviations in headings

If a command or statement has an abbreviation associated with it, the abbreviation is placed in parentheses following the full name of the command in the heading (e.g., LIST (L.)).

1.3 GLOSSARY AND TERMINOLOGY

adata (ATASCII Data) Any ATASCII character, excluding commas and carriage returns. (See Appendix C.)

aexp (Arithmetic Expression) Generally composed of a variable, function, constant, or two arithmetic expressions separated by an arithmetic operator. See section 2.3.2.

alphanumeric

The letters A through Z (either lower or upper case) and the digits 0 through 9.

aop (Arithmetic operator). See section 2.2.1.

Arrays and Array Variables

An array is a list of places where data can be filed for future use. Each of these places is called an element, and the whole array or any element is called an array variable. See section 2.1.2.

avar (Arithmetic Variable) A location where a numeric value is stored. Variable names may be from 1 to 120 alphanumeric characters, but must start with an alphabetic character. All characters are normalized to upper case normal (i.e., not inverse) video.

BASIC Beginner's All-purpose Symbolic Instruction Code.

Constant A constant is a value expressed as a number rather than represented by variable name. For example, in the statement $X = 100$, X is a variable and 100 is a constant.

Command String

Multiple commands (or program statements) placed on the same numbered line separated by colons.

exp Any expression, whether sexp or aexp. See section 2.3.

Expression An expression is any legal combination of variables, constants, operators, and functions used together to compute a value. Expressions can be either arithmetic, or string (See aexp and sexp).

filespec File Specification: A string expression that refers to a device such as the keyboard or to a disk file. It contains information on the type of I/O device, its number, a colon, an optional file name, and an optional filename extender. See section 5.1.

NOTE: BASIC XL allows you to omit the double quotes normally required in a literal string when the literal string is used as a filespec for any of the following commands:

DIR	LOAD	PROTECT	LVAR	RUN
ENTER	SAVE	RENAME	OPEN	XIO

CAUTION: when filespec is used this way, it must be the last thing on the program or command line. Also, DIR, LVAR, and RUN must always be the last command on the line.

- Function** A function is a subroutine built into the computer so that it can be called by the user's program. A function is NOT a statement. COS (Cosine), FRE (unused memory space), and INT (integer) are examples of functions. In many cases the value is simply assigned to a variable (stored in a variable) for later use. In other cases it may be printed out on the screen immediately. See chapter 6 for more on functions.
- Keyword** Any reserved word "legal" in the BASIC language. May be used in a statement, as a command, or for any other purpose. (See Appendix A for a list of all "reserved words" or keywords in BASIC XL.)
- lineno** (Line Number) A constant that identifies a particular program line in a deferred mode BASIC program. Must be an integer from 0 through 32767. Line numbering determines the order of program execution.
- Logical Line** A logical line consists of one to three physical lines, and is terminated either by a <RETURN> or when the maximum logical line limit is reached. Each numbered line in a BASIC program consists of one logical line when displayed on the screen.
- lop** (Logical Operator) See section 2.2.2.
- mvar** (Matrix Variable) Also called a Subscripted Variable. An element of an array or matrix. The variable name for the array or matrix as a whole may be any legal variable name. See section 2.1.2.
- Operator** Operators are used in expressions to tell the computer how it should evaluate the variables, constants, and functions in the expression. There are two types of operators -- arithmetic and logical. For more information, see section 2.2.

Physical Line

One line of characters as displayed on a TV or monitor screen.

sexp (String Expression) Can consist of a string variable, string literal (constant), or a function that returns a string value. See section 2.3.3.

String A string is a group of characters enclosed in quotation marks. "ABRACADABRA" is a string. So are "OSS IS THE BEST" and "123456789". A string is much like a numeric constant (e.g., 12.4), as it may be stored in a variable. A string variable is different in that its name must end in the character \$. See section 2.1.3.

svar (String Variable) A location where a string of characters may be stored. See 2.1.3 and 2.1.4.

var (Variable) Any variable. May be mvar, avar, or svar. See section 2.1.

Variable A variable is the name for a numerical or other quantity which may (or may not) change. Variable names may be up to 120 characters long. However, a variable name must start with an alphabetic letter, and may contain only letters and digits. See section 2.1.

1.4 Operating Modes

----- Direct Mode

Uses no line numbers and executes instruction immediately after <RETURN> key is pressed.

----- Deferred Mode

Uses line numbers and delays execution of instruction(s) until the RUN command is entered.

----- Execute Mode

Sometimes called RUN mode. After the RUN command is entered, each program line is processed and executed.

----- Memo Pad Mode

A non-programmable mode that allows the user to experiment with the keyboard or to leave messages on the screen. Nothing written while in Memo Pad mode affects the RAM-resident program.

NOTE: this mode is only available on the Atari 400 and 800.

2.1 Variables (var)

There are two basic types of variables in BASIC XL -- arithmetic variables and string variables. Also, there are three extensions to these -- arrays, matrices, and string arrays.

Arithmetic, array, and matrix variables all store numbers, and can only be used where a number is required.

String and string array variables both store character strings and can only be used where a character string is required.

There are limits to the number of variables you may use, and to the size and format of a variable name, as follows:

- 1) BASIC XL limits the user to 128 variable names. To bypass this problem, use individual elements of any array instead of having separate variable names. To clear the variable name table (possibly after an error 4), you can save your program using LIST, then type NEW, and then ENTER your program back in.
- 2) All variable names must start with an alphabetic letter, followed by either letters or digits. The name must be less than 120 characters long. All string or string array variable names must end in the '\$' (dollar sign) character.

2.1.1 Arithmetic Variables (avar)

Arithmetic variables are those which store a single number, and are the most common variables used. The following are examples of arithmetic variables:

```
X
THISISANARITHMETICVARIABLE
TEMP
CHARGE
```

Here are some examples of arithmetic variables in use:

```
100 LET X=76           :REM here's one use
200 FOR I=1 TO 100     :REM here's a second
300 PRINT X-2          :REM and a third
400 NEXT I
500 END
```

2.1.2 Array / Matrix Variables (mvar)

An array variable is a group of memory locations (called elements or subscripts of the array). In each one of these locations is a number; so, in essence, an array is simply a group of arithmetic variables which share a common name.

The manner in which you access a given element of an array is simple -- you merely give the array name followed by the element number in parentheses, as in the following examples:

```
A(3)    ARRAY(14)    NUMLIST(40)
```

The elements are numbered starting at 0, and continue through to the DIMensioned size of the array. "How do I dimension the size?" It's easy. You use the DIM statement as follows:

```
DIM A(40)           REM dimension 'A' as a 40 element
                    REM array.

DIM NUMLIST(60)     REM dimension 'NUMLIST' as a 60
                    REM element array.
```

For more information on the use of DIM, see section 2.1.5.

A matrix is similar to an array, except that it is two dimensional. This means that there are two numbers required to specify a given element: a row number, and

a column number. You can think of a matrix as a grid, with each box being one element. The following is a representation of a 5 by 5 matrix, where each of the boxes contains the subscripts used to access that box (element):

		C	O	L	U	M	N
		+-----+					
		0,0	0,1	0,2	0,3	0,4	
		+-----+					
Notice that the row	R	1,0	1,1	1,2	1,3	1,4	
number is given		+-----+					
first, followed by	O	2,0	2,1	2,2	2,3	2,4	
a comma and then		+-----+					
the column number.	W	3,0	3,1	3,2	3,3	3,4	
This is the same		+-----+					
order you would use		4,0	4,1	4,2	4,3	4,4	
to access that ele-		+-----+					
ment.		+-----+					

Dimensioning the size of a matrix is very similar to dimensioning an array, but both the row dimension and column dimension are required, e.g.:

```
DIM AMATRIX(4,4)  REM a 5 by 5 matrix; remember
                   REM that (0,0), not (1,1) is
                   REM the first element.
```

NOTE: for more information on DIM, see section 2.1.5.

When you use an element of an array or matrix, you are actually using a single number (which is what an arithmetic variable is). This means that an array or matrix element may be used wherever 'avar' can be used.

Examples:

```
X=47.4
ARRAY(7)=47.4
MATRIX(4,3)=47.4

IF ABS(X)<100 THEN...
IF ABS(ARRAY(7))<100 THEN...
IF ABS(MATRIX(4,3))<100 THEN...
```

2.1.3 String Variables (svar)

String variables are used to store literal strings of characters. A literal string of characters is simply a group of characters enclosed in double quotes:

```
"this is a literal string"  
"numbers in quotes are strings: 34344.2"
```

String variable names are just like arithmetic variable names, except that they must end with a '\$', as in the following examples:

```
STRING$  
A$
```

To dimension the size of a string variable (i.e., define how many characters it may hold), you use the DIM statement (also see 2.1.5):

```
DIM STRING$(66)  
DIM A$(10)
```

NOTE: BASIC XL will auto-dimension a string variable if you don't manually DIMension it. See 3.15 for more info on this feature.

With arrays and matrices the first element is the zeroeth, but with strings the first element is the first, e.g.:

```
DIM A$(10)  
A$="A String"
```

A\$(1)="A", and A\$(0) generates an error because the first element of a string is (1), not (0) (as in arrays and matrices).

2.1.4 String Array Variables (svar)

A string array is very similar to a normal arithmetic array (section 2.1.2), except that each element is a string, not a number.

As with string variables, a string array variable must have its name end with a '\$', and it is dimensioned using DIM. However, there are two quantities which need to be dimensioned -- the number of elements and the size of each element. The following examples show

how to do this (also see section 2.1.5):

```
DIM Strarray$(4,40)
DIM A$(10,100)
```

The first example dimensions a string array called "Strarray\$" with 4 elements. Each element is a string 40 characters long. The second example dimensions the string array "A\$" to 10 elements, with each element being 100 characters in length.

To access one of the elements of a string array you specify the element number (the first element is number 1, not 0 as in arithmetic arrays) followed by a semicolon (;). An example follows:

```
100 DIM A$(3,6)
200 A$(1;)="TEST"
300 A$(2;)="STRING"
400 A$(3;)="ARRAY"
```

2.1.5 DIM

Format: DIM svar(aexp[,aexp]) [,svar(aexp[,aexp])...]
DIM mvar(aexp[,aexp]) [,mvar(aexp[,aexp])...]

Example: DIM A(100)
DIM M(6,3)
DIM B\$(20)
DIM A\$(20,40)

A DIM statement is used to reserve a certain number of locations in memory for an array, matrix, string, or string array.

The first example reserves 101 locations (each of which can contain any legal numeric quantity) for an array designated A.

The second example reserves 7 rows by 4 columns for a two-dimensional array (matrix) designated M.

The third example reserves 20 bytes for the string 'B\$'.

NOTE: BASIC XL contains an auto DIMension capability for simple string variables only which you can control. For more info, see SET, section 3.15.

The fourth example reserves a string array of 20 elements, with each string element being 40 characters long.

2.2 Operators

BASIC XL has two types of operators:

- 1) Arithmetic Operators
- 2) Logical (relational) Operators

As you will see in the expressions sections, either of these two types of operators may be used in arithmetic expressions, while neither may be used in a string expression.

2.2.1 Arithmetic Operators (aop)

BASIC XL uses 7 arithmetic operators:

- + addition (also unary plus; e.g., +5)
- subtraction (also unary minus; e.g., -5)
- * multiplication
- / division
- ^ exponentiation
- & bitwise "AND" of two positive integers (both ≤ 65535)
- | bitwise "OR" of two positive integers (both ≤ 65535)
- % bitwise "EOR" of two positive integers (both ≤ 65535)

The first four are straightforward enough, but the last four require some explanation.

The "^" operator is used to raise a number to a specified power. The following examples should clarify this:

Exponent	Expanded	Result
4^2	$4*4$	16
5^3	$5*5*5$	125

	Bit A	Bit B	Bit-wise And Result
& tests two bytes bit by bit, returning a value based on this table:	1	1	1
	0	1	0
	0	0	0
	1	0	0

Example: $5 \& 39$ -- 00000101 (equals 5 decimal)
 00100111 (equals 39 decimal)
 & -----
 00000101 (result of & is 5)

	Bit A	Bit B	Bit-wise Or
1 returns a value dependent on this table:	1	1	1
	0	1	1
	0	0	0
	1	0	1

```

Example: 5 | 39 -- 00000101 (5)
              00100111 (39)
              |-----
              00100111 (result of | is 39)

```

	Bit A	Bit B	Bit-wise XOR
% returns a value dependent on this table:	1	1	0
	0	1	1
	1	0	1
	0	0	0

```

Example: 5 % 39 -- 00000101 (5)
              00100111 (39)
              %-----
              00100010 (result of % is 34)

```

2.2.2 Logical Operators (lop)

The logical operators consist of three types: relational, unary, and binary.

The rest of the binary operators are relational.

- < The first expression is less than the second expression.
- > The first expression is greater than the second.
- = The expressions are equal to each other.
- <= The first expression is less than or equal to the second.
- >= The first expression is greater than or equal to the second.
- <> The two expressions are not equal to each other.

Examples:

```

X >= 7
X <> INT(Y)

```

These operators are most frequently used in IF/THEN statements (i.e., in relational tests), but may also be used in arithmetic expressions. When used in this way, a 1 results the logical test proved true, and a 0 results if the test proved false.

The unary operator is NOT, and the binary operators are:

AND -- Logical AND
OR -- Logical OR

Examples:

10 IF A=12 AND T=0 THEN PRINT "GOOD" Both expressions
 must be true before GOOD
 is printed (that is, A
 must equal 12 and T must
 equal 0).

10 A=(C>1) AND (N<1) If both expressions true,
 A = +1; otherwise A = 0.

10 A = (C+1) OR (N-1) If either expression true,
 A = +1; otherwise A = 0.

10 A = NOT(C+1) If expression is false,
 A = +1; otherwise A = 0.

2.2.3 Operator Precedence

Operators require some kind of precedence, a defined order of evaluation, or we wouldn't know how to evaluate expressions like :

4+5*3

Is this equal to (4+5)*3 or 4+(5*3)? Without operator precedence it's impossible to tell. BASIC XL's normal precedence is very precise, as shown in the following table. The operators are listed in order of highest to lowest precedence. Operators on the same line are evaluated left to right in an expression.

()	Parentheses
< > = <= >= <>	Relational Operators when used to evaluate strings
	in arithmetic expressions
NOT + -	NOT, Unary Plus and Minus
^	Exponentiation
% ! &	bitwise EOR, OR, AND
* /	Multiplicative Operations
+ -	Additive Operations
< > = <= >= <>	Relational Operators
AND	Logical 'and'
OR	Logical 'or'

Examples showing the above precedence in use can be found in section 2.3.2.

2.3 Expressions (exp)

Expressions are constructions which obtain values from variables, constants, and functions using a specific set of operators. In BASIC XL there are two types of expressions -- arithmetic and string. Each of these is dealt with separately, but before going into the expressions themselves something needs to be said about the constant numbers used in arithmetic expressions.

2.3.1 Numbers

All numbers in BASIC XL are BCD floating point, but there are two ways to enter them -- in decimal or hexadecimal.

Decimal numbers may either be whole integers, fractions, or scientific notation. The following are examples of each:

Integers:	Fractions:	Sci. Notation:
4027	-67.254	4.33E2
-2	325.04	23.4E-14

The 'E' in the scientific notation examples stands for "exponent". The number following it is the power of ten (e.g., 4.33E2 means "4.33 multiplied by 10 squared").

Hexadecimal numbers can only be integers, and the digits must be preceded by a '\$', as in the following examples:

\$4A30	-\$0A	\$6FF
-\$E	-\$A000	\$FFFF

The maximum hexadecimal value allowed is \$FFFF.

Internal Format of Numbers:

Numbers are represented internally in 6 bytes. There is a 5 byte mantissa containing 10 BCD digits and a one byte exponent.

The most significant bit of the exponent byte gives the sign of the mantissa (0 for positive, 1 for negative). The least significant 7 bits of the exponent byte gives the exponent in excess 64 notation. Internally, the exponent represents powers of 100 (not powers of 10).

Example:

$$0.02 = 2 * 10^{-2} = 2 * 100^{-1}$$

$$\text{exponent} = -1 + 40 = 3F$$

$$0.02 = 3F\ 02\ 00\ 00\ 00\ 00$$

The implied decimal point is always to the right of the first byte. An exponent less than hex 40 indicates a number less than 1. An exponent greater than or equal to hex 40 represents a number greater than or equal to 1.

Zero is represented by a zero mantissa and a zero exponent.

In general, numbers have a 9 digit precision. For example, only the first 9 digits are significant when INPUTting a number. Internally the user can usually get 10 significant digits in the special case where there are an even number of digits to the right of the decimal point (0,2,4...).

2.3.2 Arithmetic Expressions (aexp)

Arithmetic expressions are those which evaluate to a number. Following is a list of expression elements which are considered to be numbers:

- 1) a constant number
- 2) an avar (including subscripted mvars)
- 3) a function which returns a number
- 4) two sexps compared using a relational operator

The first three are straightforward, but the fourth requires an example:

```
100 S$="ABC"
200 PRINT S$< "DEF"
300 END
```

prints out:

1

because the logical comparison of the two strings is true.

An arithmetic expression can simply be one of the above, or two or more of the above separated by

operators (either arithmetic or logical). The following are examples of arithmetic expressions, including the order of the operators' evaluation (in any) and the result:

Expression	evaluation Order	Result
-----	-----	-----
3*(4+(21/7)*2)	/,*,+,*	30
"ABC">"DEF"+7*(ASC("A"))	> ,ASC,*,+	455
X=100 : Y=2 INT(X*Y/3)	*,/,INT	66

2.3.3 String Expressions (sexp)

String expressions are much simpler than arithmetic expressions since there are fewer things they can be. The following list shows all the valid string expression possibilities:

- 1) a string constant
- 2) an svar (including subscripted string arrays)
- 3) a function which returns a string
- 4) a substring of an svar or string array

This is the first time we've seen the word "substring" used, so we need to define and to explain it.

String	Definition when Destination String	Definition when Source String
-----	-----	-----
S\$	the entire string 1 thru DIM value	from 1st thru LEN character
S\$(n)	from nth thru DIMth character	from nth thru LENGth character
S\$(n,m)	from the nth thru the mth character	from the nth thru the mth character
SA\$(e;)	same as S\$, except string is eth element of SA\$	same as S\$, except string is eth element of SA\$
SA\$(e;n)	same as S\$(n), except string is eth element of SA\$	same as S\$(n), except string is eth element of SA\$

SA\$(e;n,m)	same as S\$(n,m)	same as S\$(n,m)
	except string is	except string is
	eth element of SA\$	eth element of SA\$

A destination string is one to which something is being assigned. Any other string is a source string. In

X\$=Y\$	READ X\$	INPUT X\$
RPUT Y\$	PRINT Y\$	etc.

X\$ is the destination string, Y\$ is the source string.

An error occurs if either the first or last specified character (n and m, above), or the element number (in the case of string arrays) is outside the DIMensioned size. Also, an error occurs if the last character position given (explicitly or implicitly) is less than the first character position.

Source Example: (Assume A\$ = "VWXYZ")

1) PRINT A\$(2)	prints: WXYZ
2) PRINT A\$(3,4)	prints: XY
3) PRINT A\$(5,5)	prints: Z
4) PRINT A\$(7)	
is an error because A\$ has a length of 5.	

Destination Example: (Assume DATA "VWXYZ")

1) READ D\$	
PRINT D\$	prints: VWXYZ

Some of the commands available in BASIC XL are designed specifically to aid in quick and effective program development. The operations these commands execute are too diverse to describe in detail here, so we'll simply give their names and refer you to the section in which the particular command is discussed:

BYE	LIST	RENUM
CLR	LOMEM	RUN
CONT	LVAR	SET
DEL	NEW	STOP
DOS	NUM	TRACE
FAST	REM	TRACEOFF

3.1 BYE (B.)

Format: BYE

Example: BYE

The function of the BYE command is to exit BASIC XL and put the computer in Memo Pad mode. This allows you to experiment with the keyboard or to leave messages on the screen without disturbing any BASIC XL program in memory. To return to BASIC XL, press <SYSTEM RESET>.

3.2 CLR

Format: CLR

Example: 200 CLR

This command clears the memory of all previously dimensioned strings, arrays, and matrices so the memory and variable names can be used for other purposes. It also clears the values stored in undimensioned variables. If a matrix, string, or array is needed after a CLR command, it must be redimensioned with a DIM command.

3.3 CONT (CON.)

Format: CONT

Example: CONT
 100 CONT

In direct mode, this command resumes program execution after a STOP statement, a <BREAK> key abort, or any stop caused by an error.

CAUTION: Execution resumes on the line following the halt, so any statements following the halt (and on the same line as the halt) will not be executed.

In deferred mode, CONT may be used for error trap handling.

Example:
 10 TRAP 100
 20 OPEN #1,12,0,"D:X"
 30
 ..
 ..
 100 IF ERR(0)=170 THEN
 OPEN #1,8,0,"D:X":CONT

In line 20 we attempt to open a file for updating. If the file does not exist, a trap to line 100 occurs. If the "FILE NOT FOUND" error occurred, the file is opened for output (and thus created) and execution continues at line 30 via "CONT".

3.4 DEL

Format: DEL line[,line]

Example: DEL 1000,1999

DEL deletes program lines currently in memory. If two line numbers are given (as in the example), all lines between the two numbers (inclusive) are deleted. A single line number deletes a single line.

Example:
 100 DEL 1000,1999
 110 SET 9,1:TRAP 1000
 120 ENTER "D:OVERLAY1"
 1000 REM These lines are deleted by line 100.
 1010 REM Presumably they will be overlaid by
 1998 REM the program ENTERED in line 120.
 1999 REM See 'ENTER' and 'SET' for more info.

3.5 DOS

Format: DOS

Example: DOS

The DOS command is used to go from BASIC XL to the Disk Operating System (DOS). If the Disk Operating System has not been booted into memory, the computer will go into Memo Pad mode and the user must press <SYSTEM RESET> to return to Direct mode. If the Disk Operating System has been booted, control is given to DOS. To return to BASIC XL, press 'CAR' <RETURN> for OS/A+ or DOS XL, or press 'B' <RETURN> for Atari DOS.

NOTE: The command CP is exactly equivalent to DOS.

DOS is usually used in Direct mode; however, it may be used in a program. For more details on this, see your DOS manual.

3.6 FAST

Format: [lineno] FAST

Example: FAST
100 FAST

During normal program execution BASIC XL must search (from the beginning) for a specified line number whenever it encounters a GOTO, GOSUB, FOR, or WHILE (this is how most of the other BASICs do it too). However, you can change this by using the FAST command.

When BASIC XL sees 'FAST', it does a precompile of the program currently in memory. During the precompile BASIC XL changes every line number to the address of that line in memory. Now, when a GOTO, GOSUB, FOR, or WHILE is executed, no line number search is needed, since BASIC XL can simply jump right to the specified line's address.

NOTE: if the lineno used in the GOTO or GOSUB is not a constant (i.e., is a variable or an expression), then that lineno will not be affected by FAST, and so will RUN at normal speed.

3.7 LIST (L.)

Format: LIST [lineno [, lineno]
LIST ["filespec"[,lineno [, lineno]]]

Examples:

```
LIST
LIST 10
LIST 10,100
LIST 10,
LIST "P:"
LIST "D:DEMO.LST"
LIST "P:",20,100
```

LIST causes the program currently in memory to be displayed. You can display a single line by giving the line number after the 'LIST', or display a group of lines by giving the starting line number and ending line number (separated by a comma) after the 'LIST'.

If you give the starting line number, a comma, and no end address, the ending line number is assumed to be the last line in the program.

If no line number(s) is given, the entire program is displayed.

You can also redirect the display to a file by entering the filespec enclosed in double quotes immediately after the 'LIST'. You can then add any of the line number specifications described above to list only what you want to that file.

LIST can be used in Deferred mode as part of an error trapping routine (See TRAP in Section 4).

NOTE: the quotes around the filespec are required for LIST, unless of course a string variable is used.

3.8 LOMEM

Format: LOMEM addr

Example: LOMEM DPEEK(128)+1024

This command is used to reserve space below the normal program space. You could then use this space for screen display information or assembly language routines. The usefulness of this may be limited, though, since there are other more usable reserved areas available.

CAUTION: LOMEM wipes out any user program currently in memory.

3.9 LVAR (LV.)

Format: LVAR [filespec]

Example: LVAR P:

This statement will list (to any file) all variables currently in use. Each variable is followed by a list of the lines on which that variable is used. The example above will list the variables to the printer. If no filespec is used then LVAR lists to the screen.

NOTE: strings are denoted by a trailing '\$', arrays by a trailing '('.

WARNING: LVAR must be the last (or only) command on a line.

3.10 NEW

Format: NEW

Example: NEW

This command erases the program stored in RAM. Therefore, before typing NEW, either SAVE or CSAVE any programs to be recovered and used later. NEW clears BASIC's internal symbol table so that no arrays (See Section 8) or strings (See Section 7) are defined. NEW is normally used in Direct mode but is sometimes useful in deferred mode as an alternative to END.

3.11 NUM

Format: NUM [start][,increment]

Example: NUM
NUM 50
NUM ,1
NUM 50,1

The NUM command enables BASIC XL's automatic line numbering facility. This facility can increase your program entry speed because it puts in the program line numbers for you.

If no start or increment is given (first example), NUM will start numbering from the last line number currently in the program in increments of 10. If there

is no current program, NUM will start with line number 10.

If the starting line number alone is given (second example), NUM will start numbering from that line number in increments of 10.

If the increment alone is given (third example), NUM will start numbering from the last line currently in the program, incrementing by the number you gave it as an increment.

If both the starting line number and the increment are given (last example), NUM will start numbering from the given line number and increment by the given increment value.

Three things cause the automatic line numbering to stop:

- 1) If you press <RETURN> immediately following the line number.
- 2) If a syntax or similar error is encountered on a program line you type in.
- 3) If the next automatic line number is the same as a line number already in the program. This keeps you from overwriting previously written parts of your program.

NOTE: If the starting line number you give already exists, then the automatic line numbering will not begin.

3.12 REM (R.)

Format: REM text

Examples: 10 REM ROUTINE TO CALCULATE X
20 GOSUB 300 : REM Find Totals

REM stands for "remark" and is used to put comments into a program. This command and the text following it on the same line are ignored by the computer. However, it is included in a LIST along with the other numbered lines. Since all characters following a REM are treated as part of the REMark, no statements following it (on the same logical line) will be executed.

3.13 RENUM

Format: RENUM [start][,increment]

Examples: RENUM
 RENUM 100
 RENUM ,30
 RENUM 1000,5

RENUM rennumbers the entire program as it currently resides in memory. The first line in memory is given the line number specified by 'start', and each subsequent line number is one 'increment' greater than the last.

All line number references (e.g., in GOTO, GOSUB, etc.) are also renumbered IF the line numbers are absolute numbers. Line number expressions (e.g., GOTO 1000+10*INDEX) will NOT be renumbered.

If no 'start' line number is given, RENUM assumes a starting line number of 10. If no 'increment' is given, RENUM will renumber lines in increments of 10. (That is, just typing 'RENUM' is equivalent to typing 'RENUM 10,10'.)

As noted in the examples above, both start and increment are separately optional.

WARNING: If you use LIST in deferred mode (i.e., in a program) the lineno values you want to list will not be RENUMbered.

WARNING: RENUM will not renumber absolute linenos after a lineno expressed as an expression. Example:

ON X GOSUB 100,3*Y,200

In this example 100 will be RENUMbered, but 200 will not, since it follows a lineno expressed as an expression (3*Y).

3.14 RUN

Format: RUN [filespec]

Examples: RUN
 RUN D:MENUE

This command causes the computer to begin executing a program. If no filespec is specified, the current RAM-resident program is executed. If a filespec is included, the computer retrieves the tokenized program

from the specified file, executes a FAST command (see section 3.6), and then executes the program.

Before execution begins all variables (including arrays, strings, and matrices) are set to zero, all open files (channels) are closed, and all sounds are turned off.

Unless the TRAP command is used, an error will cause the execution to halt and an error message will be displayed.

RUN can also be used in Deferred mode.

Examples: 10 PRINT "OVER AND OVER AGAIN."
20 RUN

Type RUN and press <RETURN>. To end, press <BREAK>.

To begin program execution at a point other than the first line number, type GOTO followed by the specific line number, then press <RETURN>. CAUTION: arithmetic variables, arrays, and strings are neither cleared or initialized by GOTO.

NOTE: RUN must be the last (or only) command on a line.

3.15 SET

Format: SET aexp1,aexp2

Example: 100 SET 1,5

SET is a statement which allows you to exercise control over a variety of BASIC XL system level functions. The table below summarizes the various SET table parameters (default values are given in parentheses).

aexp1		aexp2	Meaning
-----		-----	-----
0	(0)	0	-BREAK key functions normally
		1	-User hitting BREAK cause an error to occur (TRAPable)
		128	-BREAKs are ignored
1	(10)	1	-Tab stop setting for the comma in
		thru	PRINT statements.
		128	
2	(63)	0	-Prompt character for INPUT (default
		thru	is "?").
		255	

aexpl		aexp2	Meaning
----		-----	
3	(0)	0	-FOR...NEXT loops always execute at least once (ala ATARI BASIC).
		1	-FOR loops may execute zero times (ANSI standard)
4	(0)	0	-On a mutiple variable INPUT, if the user enters too few items, he is reprompted (e.g., with "?")
		1	-Instead of reprompting, a TRAPable error occurs.
5	(1)	0	-Lower case and inverse video characters remain unchanged without causing syntax errors (BASIC XL allows mixed case program entry).
		1	-For program entry ONLY, lower case letters are converted to upper case and inverse video characters are uninverted. EXCEPTION: characters between quotes remain unchanged.
			CAUTION: this conversion applies to REMarks and DATA statements also. For total compatibility with Atari BASIC, it might be best to use SET 5,0.
6	(0)	0	-Print error messages along with error numbers (for most errors)
		1	-Print only error numbers.
7	(0)	0	-Missiles (in Player / Missile Graphics), which move vertically to the edge of the screen, roll off the edge and are lost.
		1	-Missiles wraparound from top to bottom and visa versa.
8	(1)	0	-Don't push (PHA) the number of parameters to a USR call on the stack [advantage: some assembly language subroutines not expecting parameters may be called by a simple USR(addr)].
		1	-DO push the count of parameters (ATARI BASIC standard).

aexpl -----		aexp2 -----	Meaning -----
9	(0)	0	-ENTER statements return to the READY prompt level on completion.
		1	-If a TRAP is properly set, ENTER will execute a GOTO the TRAP line on end-of-entered-file.
10	(0)	0	-The four missiles act separately; that is, as four missiles.
		1	-The four missiles are grouped into a fifth player. To move this player, you need only do a PMMOVE of one of the missiles since they are all grouped together.
11	(40)	1 thru 255	-BASIC XL will DIM a string to this size if you do not use a DIM statement to otherwise dimension it.
		0	-BASIC XL works like Atari BASIC
12	(1)	0	-The program LIST formatter does not indent when you use structured statements (FOR, WHILE, etc.).
		1	-The LIST formatter does indent when you use structured statements.

NOTE: The SET parameters are reset to the system defaults on execution of a NEW statement.

Examples:

1) SET 1,4 : PRINT 1,2,3,4

The number will be printed every four columns

2) SET 2,ASC(">")

Changes the INPUT prompt from "?" to ">"

3) 100 SET 9,1 : TRAP 120
110 ENTER "D:OVERLAY.LIS"
120 REM execution continues here after
130 REM entry of the overlay

4) 100 SET 0,1 : TRAP 200
110 PRINT "HIT BREAK TO CONTINUE"
120 GOTO 110
200 REM come here via BREAK KEY

```

5) 100 SET 3,1
    110 FOR I = 1 TO 0
    120 PRINT " THIS LINE WON'T BE EXECUTED"
    130 NEXT I

```

3.16 STOP

Format: STOP

Example: 100 STOP

When the STOP command is executed in a program, BASIC XL displays the message STOPPED AT LINE lineno, terminates program execution, and returns to Direct mode. The STOP command does not close files or turn off sounds (as does END), so the program can be resumed by typing CONT <RETURN> (see section 3.3 for more info on CONT).

3.17 TRACE and TRACEOFF

Formats: TRACE
 TRACEOFF

Examples: 100 TRACE
 TRACEOFF

These statements are used to enable or disable the line number trace facility of BASIC XL. When in TRACE mode, the line number of a line about to be executed is displayed on the screen surrounded by square brackets.

Exceptions: The first line of a program does not have its number traced. The object line of a GOTO or GOSUB and the looping line of FOR or WHILE may not be traced.

NOTE: A direct statement (e.g., RUN) is TRACED as having line number 32768.

This chapter explains the commands associated with loops, conditional and unconditional branches, error traps, and subroutines. It also explains the means of accessing data and the optional command used for defining variables.

The following commands are described in this chapter:

Assignment Statement	LET
END	MOVE
FOR...TO...STEP/NEXT	ON...GOTO/GOSUB
GOSUB...RETURN	POP
GOTO	RESTORE
IF...THEN	TRAP
IF...ELSE...ENDIF	WHILE...ENDWHILE

4.1 Assignment Statement

Format: avar=aexp
 mvar(aexp)=aexp
 svar(aexp;)=sexp [,sexp...]
 svar=sexp [,sexp...]

Example: X=9
 I=X+7*9
 ARRAY(7)=23.75
 A\$(4;)="A STRING ARRAY ELEMENT"
 S\$="THIS IS A STRING"
 M\$="CONCATENATED"
 C\$=S\$," WHICH IS ",M\$

The assignment statement is used to assign a value to a variable, and can be used with arithmetic, matrix (array), or string variables (including string arrays).

The first and second examples given simply equate an avar to an aexp. If you insert a 'PRINT I' statement after the second example, 72 (the value of I) will be printed. The third equates one element of a mvar to an aexp.

The fourth example is somewhat more complicated; it equates one element of a string array to a sexp (in this case a string constant).

The fifth and sixth examples equate svars to sexps.

String concatenation may be accomplished via the form shown in the last example above. Note that

```
A$=B$,C$
```

is exactly equivalent to

```
A$=B$  
A$(LEN(A$)+1)=C$
```

From this you can see that C\$ in the last example is equal to "THIS IS A STRING WHICH IS CONCATENATED".

Here is another example:

```
100 DIM A$(100),B$(100)  
200 A$="123"  
300 B$="ABC"  
400 A$=A$,B$,A$  
500 REM At this point A$ = "123ABC123ABC"  
600 A$(4,9)="X",STR$(3*7),"X"  
700 REM At this point, A$="123X21X23ABC"  
800 A$(7)=A$(1,3)  
900 REM Finally, A$="123X21123"
```

NOTE: for more information on variables and expressions, see chapter 2.

4.2 END

Format: END

Example: 1000 END

This command is used to terminate the execution of a program. In addition to this, it also closes all files and turns off any sounds. It does not change the GRAPHICS mode, however. END is not required in most programs because BASIC XL automatically closes all files and turns off any sounds after the last program line has executed.

If you have any subroutines following the main program you should put an END at the end of the main program; otherwise the subroutines will be executed as part of the main program.

END may also be used in Direct mode to close files and turn off sounds.

4.3 FOR(F)...TO...STEP / NEXT(N.)

Format: FOR avar = aexp1 TO aexp2 [STEP aexp3]
 NEXT avar

Examples: FOR X = 1 TO 10
 NEXT X

 FOR Y = 10 to 20 STEP 2
 NEXT Y

 FOR INDEX = Z TO 100 * Z
 NEXT INDEX

The FOR statement is used to repeat a group of statements a specified number of times. It does this by initializing the loop variable (avar) to the value of aexp1. Each time the NEXT avar statement is encountered, the loop variable is incremented by the amount specified by aexp3 in the 'STEP' option. aexp3 can be either positive or negative, either a fraction or a whole number. If the 'STEP' option is not used, the loop increments by one. When the loop completes the limit as defined by aexp2, it stops and the program proceeds to the statement immediately following the NEXT statement.

FOR loops can be nested, one within another. In this case, the innermost loop is completed before returning to the outer loop. The following example illustrates a nested loop program.

```
10 FOR X=1 TO 3           : REM START OF OUTER LOOP
20 PRINT "OUTER LOOP"
30 Z=0
40 Z=Z+2
50 FOR Y=1 TO 5 STEP Z     : REM START OF INNER LOOP
60 PRINT "    INNER LOOP"
70 NEXT Y                 : REM END OF INNER LOOP
80 NEXT X                 : REM END OF OUTER LOOP
90 END
```

The outer loop will complete three passes (X = 1 to 3). However, before this first loop reaches its NEXT X statement, the program gives control to the inner loop. Note that the NEXT statement for the inner loop must precede the NEXT statement for the outer loop. In the example, the inner loop's number of passes is determined by the STEP statement (STEP Z). In this case, Z has been defined as 0, then redefined as Z+2. Using this data, the computer must complete three passes through the inner loop before returning to the

outer loop. The following is the output of the program when it is RUN:

```
OUTER LOOP
  INNER LOOP
  INNER LOOP
  INNER LOOP
OUTER LOOP
  INNER LOOP
  INNER LOOP
  INNER LOOP
OUTER LOOP
  INNER LOOP
  INNER LOOP
  INNER LOOP
```

The return addresses for the loops are placed in a special group of memory addresses referred to as a stack. The information is "pushed" on the stack and when used, the information is "popped" off the stack (see POP).

4.4 GOSUB (GOS.) / RETURN (RET.)

Format: GOSUB linenol
 linenol
 :
 :
 lineno2 RETURN

Example: 100 GOSUB 2000
 2000 PRINT "SUBROUTINE"
 2010 FOR X=1 TO 10
 2020 PRINT X,X*X
 2030 NEXT X
 2040 RETURN

A subroutine is a program or routine used to compute a certain value, etc. It is generally used when an operation must be executed several times within a program sequence using the same or different values. This command allows the user to "call" the subroutine, if necessary. The last line of the subroutine must contain a RETURN statement. The RETURN statement goes back to the physical line following the GOSUB statement.

Generally, a subroutine can do anything that can be done in a program. It is used to save memory and program-entering time, and to make programs easier to read and debug.

Like the preceding FOR/NEXT command, the GOSUB/RETURN command uses a stack for its return address. If the subroutine is not allowed to complete normally; e.g., a GOTO lineno before a RETURN, the GOSUB address must be "popped" off the stack (see POP) or it could cause future errors.

To prevent accidental triggering of a subroutine (which normally follows the main program), place an END statement preceding the subroutine. The following program demonstrates the use of subroutines.

```

10 PRINT CHR$(125) :REM this clears the screen
20 REM EXAMPLE USE OF GOSUB/RETURN
30 X=100
40 GOSUB 1000
50 X=120
60 GOSUB 1000
70 X=50
80 GOSUB 1000
90 END
1000 Y=3*X
1010 X=X+Y
1020 PRINT X,Y
1030 RETURN

```

In the above program, the subroutine, beginning at line 1000, is called three times to compute and print out different values of X and Y. Below are the results of executing this program.

400	300
480	360
200	150

4.5 GOTO (G.)

Format: [lineno] GOTO aexp

Examples: 100 GOTO 50
500 GOTO (X + Y)

The GOTO command is an unconditional branch statement just like the GOSUB command. They both immediately transfer program control to a target line number or arbitrary expression. However, You cannot RETURN from a GOTO, as you can with a GOSUB. If the target line number is non-existent, an error results. Any GOTO statement that branches to a preceding line may result in an "endless" loop. Statements following a GOTO statement will not be executed. Note that a conditional branching statement (see IF/THEN) can be

1

Figure 1

1

4.6 IF/THEN

Format: IF aexp THEN lineno
IF aexp THEN statement [:statement...]

Examples: IF X = 100 THEN 150
IF A\$ = "ATARI" THEN 200
IF AA = 145 and BB = 1 THEN PRINT AA,BB
IF X = 100 THEN X = 0

See also IF...ELSE...ENDIF discussion in the following section.

The IF/THEN statement is a conditional branch statement. This type of branch occurs only if certain conditions are met. These conditions may be either arithmetical or logical. If the aexp following the IF statement is true and/or non-zero, the program executes the THEN part of the statement. If, however, the aexp is false and/or zero, the rest of the statement is ignored and program control passes to the next numbered line.

In the format, IF aexp THEN lineno

lineno must be a constant (not an expression) specifying the line number to go to if the expression is true. If several statements occur after the THEN, separated by colons, then they will be executed if and only if the expression is true. Several IF statements may be nested on the same line. For example:

```
100 IF X=5 THEN IF Y=3 THEN R=9: GOTO 200
```

The statements R=9 : GOTO 200 will be executed only if X=5 and Y=3. The statement Y=3 will be executed if X=5. The following program demonstrates the IF/THEN statement:

```
100 GRAPHICS 0 : PRINT
110 PRINT ,, "IF DEMO"
120 PRINT : PRINT "ENTER A"; : INPUT A
130 IF A=1 THEN 150 ; REM Multiple Statements
    here will never be executed!!!
140 PRINT : PRINT "A IS NOT 1, "EXECUTION
    CONTINUES HERE WHEN EXPRESSION IS FALSE."
150 IF A=1 THEN PRINT : PRINT "A=1?" : PRINT
    "YES, IT IS REALLY 1." ; REM Multiple statements
    here will be executed only if A=1!!!
160 PRINT : PRINT "EXECUTION CONTINUES HERE IF
    A <> 1 OR AFTER 'YES, IT IS REALLY 1' IS DISP
    LAYED."
60 GOTO 10
```

Output of the above program is:

IF DEMO

ENTER A ? (entered 2)
A IS NOT 1. EXECUTION CONTINUES HERE WHEN
THE EXPRESSION IS FALSE.
EXECUTION CONTINUES HERE IF A <> 1 OR AFTER
'YES', IT IS REALLY 1' IS DISPLAYED.

ENTER A ? (entered 1)

A=1
YES, IT IS REALLY 1.
EXECUTION CONTINUES HERE IF A <> 1 OR AFTER
'YES, IT IS REALLY 1' IS DISPLAYED.

4.7 IF...ELSE...ENDIF

Format: IF aexp: statement [:statements...]
[ELSE: [statements...]]
ENDIF

Examples: 200 IF A>100:PRINT "TOO BIG"
210 A=100
220 ELSE:PRINT "A-OK"
230 ENDIF

1000 IF A>C : B=A : ELSE : B=C : ENDIF

BASIC XL makes available an exceptionally powerful conditional capability via IF...ELSE...ENDIF.

In the format given, if the expression is TRUE (evaluates as non-zero) then all statements between the following colon and the corresponding ELSE (if it exists) or ENDIF (if no ELSE exists) are executed; if ELSE exists, the statements between it and ENDIF are skipped.

If the expression is FALSE (evaluates to zero), then the statements (if any) between the colon and ELSE are skipped and those between ELSE and ENDIF are executed. If no ELSE exists, all statements through the ENDIF are skipped.

CAUTION: The colon following the aexp IS REQUIRED and MUST be followed by a statement. The word THEN is NOT ALLOWED in this format.

There may be any number (including zero) of statements and lines between the colon and the ELSE and between the ELSE and the ENDIF.

The second example above sets B to the larger of the values of A and C.

This IF structure may also be nested, as follows:

```
100 IF A>B : REM SO FAR A IS BIGGER
110   IF A>C : PRINT "A BIGGEST"
120   ELSE : PRINT "C BIGGEST"
130   ENDIF
140 ELSE
150   IF B>C : PRINT "B BIGGEST"
160   ELSE : PRINT "C BIGGEST"
170   ENDIF
180 ENDIF
```

4.8 LET

Format: [LET] <assignment statement>

Example: LET GOTO=3.5
 LET LETTER\$="a"
 LET AND\$="*",AS,A\$,A\$,A\$,A\$

LET is an optional keyword which allows you to assign a value to a variable name which starts with or is identical to a reserved name. For example:

```
10 LET GOSUBBER = 5
20 LET PRINT = 7
30 LET LET = PRINT + GOSUBBER
40 PRINT PRINT,LET,GOSUBBER
```

will print out:

7 12 5

There are a few keywords which CANNOT be used as variable names through the use of LET, including any function name and the NOT unary operator.

Here is an example of what will happen if you try to use one of the above as a variable name:

```
10 CSHARP = 37
20 LET NOTE = CSHARP
30 PRINT NOTE
```

will print out: 1

If you LIST the program out you will see why. It lists "30 PRINT NOTE" as

30 PRINT NOT E

because the interpreter does not allow NOT to start a variable name.

4.9 MOVE

Format: MOVE aexp1,aexp2,aexp3

Example: MOVE \$D000, \$8000, \$400

CAUTION: be careful with this command!!

MOVE is a general purpose byte move utility which will move any number of bytes from any address to any address at assembly language speed. NO ADDRESS CHECKS ARE MADE!!

aexp1 is the starting address of the block you want to move, aexp2 is the starting address of the place where you want the block moved to, and aexp3 is the length of the block.

The sign of the third aexp (the length) determines the order in which the bytes are moved, as follows:

If the length is positive:
 (from) -> (to)
 (from+1) -> (to+1)
 ...
 (from+len-1) -> (to +len-1)

When the length is positive, the destination block can overwrite lower part of the source block.

If the length is negative:
 (from+len-1) -> (to+len-1)
 (from+len-2) -> (to+len-2)
 ...
 (from+1) -> (to +1)
 (from) -> (to)

When the length is negative, the destination block can overwrite the upper part of the source block.

```
Format:      ON aexp |GOTO | lineno [,lineno...]  
              |GOSUB|
```

```
EXAMPLES: 100 ON X GOTO 200,300,400
          100 ON A GOSUB 1000,2000
          100 ON SQR(X) GOTO 30,10,100
```

NOTE: GOSUB and GOTO may not be abbreviated when used in conjunction with ON.

These two statements are also conditional branch statements like the IF/THEN statement. However, these two are more powerful. The aexp must evaluate to a positive number which is then rounded to the nearest positive integer (whole number) value up to 255. If the resulting number is 1, then program control passes to the first lineno in the list following the GOSUB or GOTO. If the resulting number is 2, program control passes to the second lineno in the list, and so on.

If the resulting number is 0 or is greater than the number of lines in the list, the conditions are not met and program control passes to the next statement which may or may not be located on the same line. With ON/GOSUB, the selected subroutine is executed and then program control passes to the statement following the ON/GOSUB.

The following routine demonstrates the ON/GOTO statement:

```

10 X=X+1
20 ON X GOTO 100,200,300,400,500
30 IF X>5 THEN PRINT "COMPLETE.":END
40 GOTO 10
50 END
100 PRINT "NOW WORKING AT LINE 100":GOTO 10
200 PRINT "NOW WORKING AT LINE 200":GOTO 10
300 PRINT "NOW WORKING AT LINE 300":GOTO 10
400 PRINT "NOW WORKING AT LINE 400":GOTO 10
500 PRINT "NOW WORKING AT LINE 500":GOTO 10

```

When the program is executed, it looks like the following:

```

NOW WORKING AT LINE 100
NOW WORKING AT LINE 200
NOW WORKING AT LINE 300
NOW WORKING AT LINE 400
NOW WORKING AT LINE 500
COMPLETE.

```

4.11 POP

Format: POP

Example: 1000 POP

In the description of the FOR/NEXT statement, the stack was defined as a group of memory addresses reserved for return addresses. The top entry in the stack controls the number of loops to be executed and the RETURN target line for a GOSUB. If a subroutine is not terminated by a RETURN statement, the top memory location of the stack is still loaded with some numbers. If another GOSUB is executed, that top location needs to be cleared. To prepare the stack for a new GOSUB, use a POP to clear the data from the top location in the stack.

The POP command could be used in the following ways:

1) In a FOR or WHILE statement, when you wish jump out of the loop before it has executed its specified number of times (e.g., if you are searching through a lot of data for a specific item, you can leave the loop early by POPping the stack, and then using GOTO to continue execution after the NEXT). Example:

```
10 FLAG = 1
20 WHILE FLAG
30 INPUT FLAG
40 IF FLAG < 0 THEN POP : GOTO 70
50 PRINT "IN THE WHILE LOOP"
60 ENDWHILE
70 END
```

2) After a subroutine (GOSUB) which does not give control back to the main program through the use of a RETURN. The following example illustrates this instance:

```
100 REM POP Demo
110 N = 1 : GOSUB 800
120 N = 2 : GOSUB 800
130 END
800 PRINT "At Line 800"
810 GOSUB 900
820 PRINT "At Line 820"
830 RETURN
900 PRINT "At Line 900"
910 IF N = 2 THEN POP
920 RETURN
```

4.12 RESTORE (RES.)

Format: RESTORE [aexp]

Example: 100 RESTORE
 220 RESTORE X+2

BASIC XL contains an internal "pointer" that keeps track of the DATA statement item to be read next. When used without the optional aexp, the RESTORE statement resets that pointer to the first DATA item in the program. When used with the optional aexp, the RESTORE statement sets the pointer to the first DATA item on the line specified by the value of the aexp.

This statement permits repetitive use of the same data, as shown in the following example:

```
10 FOR N=2 TO 1 STEP -1
20  RESTORE 80+N
30  READ A, B
40  M=A+B
50  PRINT "TOTAL EQUALS ";M
60 NEXT N
70 END
81 DATA 30,15
82 DATA 10,20
```

On the first pass through the loop, A will be 10 and B will be 20 so the total in line 50 will print: TOTAL EQUALS 30, but on the second pass, A will equal 30 and B will equal 15, so the PRINT statement in line 50 will display: TOTAL EQUALS 45.

4.13 TRAP (T.)

Format: TRAP aexp

Example: 100 TRAP 120

The TRAP statement is used to direct the program to a specified line number if an error is detected. Without a TRAP statement, the program stops executing when an error is encountered and displays an error message on the screen.

TRAP works for any error that may occur after it (the TRAP statement) has been executed, but once an error has been detected and trapped, it is necessary to reset the error trapping with another TRAP statement. This TRAP statement should be placed at the beginning of the section of code that handles input from the keyboard so that the TRAP is reset after each error.

You can find out the error number using the ERR function with an argument of 0, and find out the lineno on which the error occurred by using the ERR function with an argument of 1 (see section 6.6.4 for a more detailed discussion of ERR).

Alternatively, PEEK(195) will give you the error number, and DPEEK(186) will give you the number of the line where the error occurred.

A TRAP may be disabled by executing a TRAP statement with an aexp whose value is zero (0), or between 32768 and 65535 (e.g., TRAP 40000).

4.14 WHILE...ENDWHILE

Format: WHILE aexp : <statements> : ENDWHILE

Example: 100 A=3
 110 WHILE A: PRINT A
 120 A=A-1 : ENDWHILE

With WHILE, the BASIC XL user has yet another powerful control structure available. So long as the aexp of WHILE remains non-zero, all statements between WHILE and ENDWHILE are executed.

Example: WHILE 1 :
 The loop executes forever

Example: WHILE 0 :
 The loop will never execute

CAUTION: Do not GOTO out of a WHILE loop or a nesting error will likely result (unless you use POP first).

NOTE: The aexp is only tested at the top of each passage through the loop.

This chapter describes the input/output devices and how data is moved between them. The commands explained in this chapter are those that allow access to the input/output devices. The input commands are those associated with putting data into RAM and the devices geared for accepting input. The output commands are those associated with retrieving data from RAM and the devices geared for generating output.

The commands described in this chapter are:

BGET	DIR	LPRINT	PROTECT	SAVE
BPUT	ENTER	NOTE	PUT	STATUS
CLOAD	ERASE	OPEN	READ	TAB
CLOSE	GET	POINT	RENAME	UNPROTECT
CSAVE	INPUT	PRINT	RGET	XIO
DATA	LOAD	PRINT USING	RPUT	

5.1 Comments and Notations

The Atari Personal Computer considers everything except the guts of the computer (i.e. the RAM, ROM, and processing chips) to be external devices. Some of these devices come with the computer, for example the Keyboard and the Screen Editor. Some of the other devices are Disk Drive, Program Recorder (cassette), and Printer. The following is a list of the devices, ordered according to the name used as 'filespec' in the BASIC XL commands:

C: The Program Recorder -- handles both Input and Output. You can use the recorder as either an input or output device, but never as both simultaneously.

D1: - D8: Disk Drive(s) -- handles both Input and Output. Unlike C:, disk drives can be used for input and output simultaneously. Floppy disks are organized into a group of files, so you are required to specify a file name along with the device name (see your DOS manual for more information).

NOTE: if you use D: without a drive number, D1: is assumed.

- E: Screen Editor -- handles both Input and Output. The screen editor simulates a text editor/word processor using the keyboard as input and the display (TV or Monitor) as output. This is the editor you use when typing in a BASIC XL program. When you specify no channel while doing I/O, E: is used because the channel defaults to 0, which is the channel BASIC XL opens for E:.
- K: Keyboard -- handles Input only. This allows you access to the keyboard without using E:.
- P: Parallel Port on the 850 Module -- handles Output only. Usually P: is used for a parallel printer, so it has come to mean 'Printer' as well as 'Parallel Port'.
- R1: - R4: The four RS-232 Serial Ports on the Atari 850 Interface -- handle both Input and Output. These devices enable the Atari system to interface to RS-232 compatible serial devices like terminals, plotters, and modems.
- NOTE: if you use R: without a device number, R1: is assumed.
- S: The Screen Display (either TV or Monitor) -- handles both Input and Output. This device allows you to do I/O of either characters or graphics points with the screen display. The cursor is used to address a screen position.

Each of these devices is used for I/O of some type, although only a few of them can do both Input and Output (you wouldn't want to input data from a Printer). Because the way in which they work is different, each device has to tell the computer how it operates. This is done through the use of a device handler. A device handler for a given device gives information on how the computer should input and output data for that device.

One of the sub-systems in the computer is the Central Input Output processor (CIO). It's CIO's job to find out if the device you specify exists, and then look up I/O information in that device's handler. This makes it easy for you, since you don't need to know anything about given handler.

To let CIO know that a device exists (i.e., is available for I/O) you need to OPEN (section 5.16) the device on one of the CIO's eight channels (numbered

8-7). When you then want to do I/O involving the OPENed device, you use the channel number instead of the device name.

When you see 'filespec' in the following sections, it refers simply to the device (and file name in the case of D:) in a character string. The string may either be a literal string (i.e., enclosed in quotes), a string of characters (not in quotes), or a string variable.

IF IOCB #7 is in use, it will prevent LPRINT or some of the other BASIC I/O statements from being performed.

```
+-----+
| In the examples in the following sections, you will |
| often see the wildcard characters * and ? in the |
| filespec. For information on the use of these, see |
| your DOS manual. |
+-----+
```

5.2 BGET

Format: BGET #channel, aexp1, aexp2

Example: (see below)

BGET gets "aexp2" bytes from the device or file specified by "channel" and stores them at address "aexp1".

NOTE: The address may be a memory address. For example, a screen full of data could be displayed in this manner. Or the address may be the address of a string. In this case BGET does not change the length of the string; this is the user's responsibility.

Example: 10 DIM A\$(1025)
 20 BGET #5,ADR(A\$),1024
 30 A\$(1025) = CHR\$(0)

This program segment will get 1024 bytes from the file or device associated with file number 5 and store it in A\$. Statement 30 sets the length of A\$ to 1025.

NOTE: No error checking is done on the address or length so care must be taken when using this statement.

For another example using BGET, see section 5.31.

5.3 BPUT

Format: BPUT #channel, aexp1, aexp2

Example: BPUT #5, ADR(A\$), LEN(A\$)

BPUT outputs a block of data to the device or file specified by "channel". The block of data starts at address "aexp1" for a length of "aexp2".

NOTE: The address may be a memory address. For example, the whole screen might be saved. Or the address may be the address of a string obtained using the ADR function.

The example above writes the block of data contained in the string A\$ to the file or device associated with channel number 5.

NOTE: nothing is written to the file which indicates the length of the data written. You are advised to write fixed-length data to make the rereading process simpler.

5.4 CLOAD

Format: CLOAD

Examples: CLOAD
 100 CLOAD

This command can be used in either Direct or Deferred mode to load a program from cassette tape into RAM for execution. On entering CLOAD, one bell rings to indicate that the PLAY button needs to be pressed followed by <RETURN>. However, do not press PLAY until the tape has been positioned. Specific instructions for CLOADing a program are contained in the ATARI 410 Program Recorder Manual.

5.5 CLOSE (CL.)

Format: CLOSE #channel

Example: CLOSE #4
 100 CLOSE #1

The CLOSE command is used to close a CIO channel which has been previously OPENed to allow I/O with some device. After you CLOSE a channel, you can then reOPEN it to some other device, and thus associate that channel number with a different device.

NOTE: you should CLOSE all channels you have OPENed when you are finished using them.

NOTE: END will also close all channels (i.e., files).

5.6 CSAVE (CS.)

Format: CSAVE

Example: CSAVE
100 CSAVE
100 CS.

This command is usually used in Direct mode to save a RAM-resident program onto cassette tape. CSAVE saves the tokenized version of the program. On entering CSAVE two bells ring to indicate that the PLAY and RECORD buttons must be pressed followed by <RETURN>. Do not, however, press these buttons until the tape has been positioned. It is faster to save a program using this command rather than a SAVE "C" (See SAVE) because short inter-record gaps are used.

NOTE: Tapes saved using the two commands SAVE and CSAVE are not compatible.

NOTE: Due to a flaw in the Atari OS ROMs, it may be necessary on some machines to enter an LPRINT (See LPRINT) before using CSAVE. Otherwise, CSAVE may not work properly.

For specific instructions on how to connect and operate the hardware, cue the tape, etc., see the ATARI 410 Program Recorder Manual.

5.7 DATA (D.)

Format: DATA adata [,adata]

Example: 100 DATA 12,13,14,15,16
200 DATA GEORGE, EVELYN, MIKE, BECKY
300 DATA "DATA with a comma, in quotes"

The DATA command is used in conjunction with the READ command (see section 5.22) to access elements in a data list. A DATA command may be anywhere in a program, but it must contain as many pieces of data as there are defined in the READ command; otherwise an "out of data" error is displayed on the screen.

NOTE: all characters except comma and <RETURN> are allowed. However, if you put the data in quotes, then all characters except double quote and <RETURN> are legal.

5.8 DIR

Format: DIR [filespec]

Example: DIR D:*.COM
DIR FILE\$
DIR "D2:TEST*.B*"

The DIR command is used to list the contents of a disk directory to the screen. It is very similar to the OS/A+ and DOS XL 'DIR' command. If no filespec is given, all files on D1: are displayed.

The first example will display all files on D1: which end with .COM.

The second example shows a string variable being used as a filespec. This is legal, but the string variable must contain a valid filespec, otherwise an error will occur.

The third example will display all files on disk Drive 2 which match TEST*.B*.

NOTE: DIR must be used as the last (or only) command on a line.

5.9 ENTER (E.)

Format: ENTER filespec

Examples: ENTER "C:"
ENTER D2:DEMOPR.INS
ENTER FILE\$

The ENTER command allows you to read in a program you have saved using the LIST command, and will not work with programs which have been SAVED or CSAVED. To use this command, you simply need to give the filespec of the program.

NOTE: whereas both LOAD and CLOAD clear the old program from memory before reading in the new one, ENTER does not, and so is useful when trying to merge programs together.

ENTER can be modified using the SET command. For an example of this, see section 3.15, example 3.

5.10 ERASE

Format: ERASE filespec

Example: ERASE "D:*.BAK"
ERASE D2:TEST?.SAV

ERASE will erase any unprotected files which match the given filespec. The first example above would erase all .BAK (back-up) files on disk drive 1. The second example would erase all files matching TEST?.SAV on disk drive 2. This command is similar to the OS/A+ and DOS XL ERASE, but there are no default file specifiers.

5.11 GET

Format: GET #channel,avar

Example: 100 GET #0,X

The GET command is used to input one byte of data from an open channel. This byte of information is stored in 'avar'.

For a program example using GET, see section 5.31.

5.12 INPUT (I.)

Format: INPUT [#chan,] |avar [, (avar)...]|
 svar

Examples: 100 INPUT X
 100 INPUT N\$
 100 INPUT X,Y,Z(4)
 100 INPUT ARRSTR\$(5;)
 100 PRINT "ENTER THE VALUE OF X"
 110 INPUT X

INPUT is used to read in various data. With it you can input either one or more numbers, or a string. If you are inputting a group of numbers, the first number will go into the first avar specified, the second number into the second avar, and so on.

NOTE: In BASIC XL the avar may be an array element, and the svar may be a string array element.

If a channel number is specified (followed by a comma), then no "?" prompt is given. This allows you to create your own prompts, as shown in the following example:

```
100 PRINT "command>> ";  
110 INPUT #0, COMMAND$
```

The statement 'INPUT #0, COMMAND\$' inputs a string from channel 0 (E:), without printing out a '?' first.

NOTE: if the user's sole response to an INPUT prompt is <CTRL>C <RETURN>, a special error (number 27) will be issued by INPUT. This can be useful in data entry manipulations.

If an INPUT request is made for more than one numeric variable, the user may respond with several values separated by commas or may type in single number on each line, followed by <RETURN>.

In the latter case, BASIC XL will prompt with a double question mark to indicate that more input is needed. When a string is requested, it must be typed on a line by itself (or, if combined with numeric input, as the last item on the line).

OSS strongly recommends that:

- 1) no more than one variable be used on each INPUT line.
- 2) INPUT and PRINT should not be used for disk data file access (RGET and RPUT are suggested instead).

5.12.1 Advanced use of INPUT

Format: INPUT "string", var [,var...]

Example: 100 INPUT "3 VALUES>> ",V(1),V(2),V(3)

BASIC XL allows you to include a prompt with the INPUT command to produce easier to use programs, without having to use the ";" option mentioned in the previous section. The string given in the above format ALWAYS replaces the default "?" prompt.

NOTE: no channel number may be used when the literal prompt is present.

NOTE: in the example above, if the user typed in only a single value followed by a <RETURN>, he would be reprompted by BASIC XL with a "??", but see chapter 3 for variations available via SET.

5.13 LOAD (LO.)

Format: LOAD filespec

Example: LOAD D1:GAME1.BXL
 100 LOAD "C:"

LOAD allows you to load the SAVED version of a program into memory from any device. It will not work properly with programs saved using LIST or CSAVE, as they have their own loading commands (see ENTER and CLOAD).

5.14 LPRINT (LP.)

Format: LPRINT [exp][|;| exp...]
 |,|

Example: LPRINT "PROGRAM TO CALCULATE X"

This statement causes the computer to print data on the line printer rather than on the screen. It can be used in either Direct or Deferred mode, and requires no device specifier, no OPEN, or no CLOSE statement.

NOTE: the semicolon and comma options are discussed in section 5.18, PRINT.

CAUTION: with most printers, LPRINT cannot successfully be used with a trailing comma or semicolon. If advanced printing capabilities are required, we recommend using PRINT # on a channel previously OPENed to the printer (P:).

5.15 NOTE (NO.)

Format: NOTE #chan,avar,avar

Example: 100 NOTE #1,X,Y

This command is used to store the current disk sector number in the first avar and the current byte number within the sector in the second avar. This is the current read or write position in the specified file where the next byte to be read or written is located.

5.16 OPEN (O.)

Format: OPEN #chan,aexpl,aeaxp2,filespec

Example: 100 OPEN #2,8,0,"C:"
100 A\$ = "D1:TEST.DAT"
110 OPEN #2,8,0,A\$

As mentioned in section 5.1, a device must be OPENed on a specific channel before it can be accessed. This "opening" process links a specific channel to the appropriate device handler, initializes any CIO-related control variables, and passes any device-specific options to the device handler.

The parameters for the OPEN command are defined as follows:

chan This is the number of the channel which you want to associate with the the device 'filespec'. Also, this is the number you use when you later want to do I/O involving the specified device (using INPUT, PRINT, etc.).

aexpl This is the I/O mode you want to associate with the above channel. The number codes are described in the following table:

aexpl	Meaning
-----	-----
4	Input only
6	Read disk directory only
8	Output only
9	Output Append. This mode allows you to append to already existing disk files.
12	Input and Output

aeaxp2 Device-dependent auxiliary code. See your device manual to see if it uses this number. If not, use a zero.

filespec The device (and file name, if required) you want to be associated with the specified channel.

5.17 POINT(P.)

Format: POINT #chan,avar,avar

Example: 100 POINT #2,A,B

This command is used when reading a file into RAM. The first avar specifies the sector number and the second avar specifies the byte within that sector where the next byte will be read or written. Essentially, it moves a software-controlled pointer to the specified location in the file. This gives the user "random" access to the data stored on a disk file. The POINT and NOTE commands are discussed in more detail in your DOS Manual.

5.18 PRINT (PR or ?)

Format: PRINT [#chan] [|;| exp...] |;|
 |,| |,|

Examples: PRINT

```
PRINT X,Y,Z, A$  
100 PRINT "THE VALUE OF X IS ";X  
100 PRINT "COMMAS", "CAUSE", "COLUMNS"  
100 PRINT #3,A$  
100 PRINT #0;"$";HEX(X);" IS ";X
```

The PRINT command is used in either Direct or Deferred mode to output data. In Direct mode, this command prints whatever information is contained between the quotation marks exactly as it appears. In the second example, PRINT X,Y,Z,A\$, the screen will display the current values of X,Y,Z, and A\$ as they appear in the RAM-resident program. In the fifth example, A\$ is PRINTed out to the device associated with channel 3.

The comma option causes tabbing to the next tab location. Several commas in a row cause several tab jumps. A semicolon causes the next aexp or sexp to be placed immediately after the preceding expression with no spacing. Therefore, in the third example a space is placed before the ending quotation mark so the value of X will not be placed immediately after the word "IS".

If no comma or semicolon is used at the end of a PRINT statement, then a <RETURN> is output and the next PRINT will start on the following line.

5.19 PRINT USING

Format: PRINT [#ch;]USING sexp,exp [,exp...]

Example: (see below)

PRINT USING allows the user to specify a format for the output to the device or file associated with "ch" (or to the screen). The format string "sexp" contains one or more format fields. Each format field tells how an expression from the expression list is to be printed. Valid format field characters are:

& * + - \$, . % ! /

Non-format characters terminate a format field and are printed as they appear.

Example 1) 100 PRINT USING "## ###X#",12,315,7

2) 100 DIM A\$(10) : A\$="## ###X#"
200 PRINT USING A\$,12,315,7

Both 1) and 2) will print

12 315X7

Where a blank separates the first two numbers and an X separates the last two.

Numeric Formats:

The format characters for numeric format fields are:

& * + - \$, .

DIGITS (# & *)

Digits are represented by:

& *

- # - Indicates fill with leading blanks
- & - Indicates fill with leading zeroes
- * - Indicated fill with leading asterisks

If the number of digits in the expression is less than the number of digits specified in the format then the digits are right justified in the field and preceded with the proper fill character.

NOTE: In all the following examples b is used to represent a blank.

Example:

Value	Format Field	Print Out
1	###	bb1
12	###	b12
123	###	123
1234	###	234
12	&&&	Ø12
12	***	*12

DECIMAL POINT(.)

A decimal point in the format field indicates that a decimal point be printed at that location in the number. All digit positions that follow the decimal point are filled with digits. If the expression contains fewer fractional digits than are indicated in the format, then zeroes are printed in the extra positions. If the expression contains more fractional digits than indicated in the format, then the expression is rounded so that the number of fractional digits is equal to the number of format positions specified.

A second decimal point is treated as a non-format character.

Example:

Value	Format Field	Print Out
123.456	###.##	123.46
4.7	###.##	bb4.7Ø
12.35	##.##.	12.35.

COMMA (,)

A comma in the format field indicates that a comma be printed at that location in the number. If the format specifies a comma be printed at a position that is preceded only by fill characters (Ø b *) then the appropriate fill character will be printed instead of the comma.

The comma is a valid format character only to the left of the decimal point. When a comma appears to the right of a decimal point, it becomes a non-format character. It terminates the format field and is printed like a non-format character.

Example:

Value	Format Field	Print Out
5216	###,###	b5,216
3	###,###	bbbbbb3
4175	**,***	*4,175
3	&&, &&&	000003
42.71	##.##,	42.71,

SIGNS (+ -)

A plus sign in a format field indicates that the sign of the number is to be printed. A minus sign indicates that a minus sign is to be printed if the number is negative and a blank if the number is positive.

Signs may be either fixed, floating or trailing.

A fixed sign must appear as the first character of a format field.

Example:

Value	Format Field	Print Out
43.7	+###.##	+b43.7
-43.7	+###.*	-b43.7
23.58	-&&&.&&	b023.58
-23.58	-&&&.&&	-023.58

Floating signs must start in the first format position and occupy all positions up to the decimal point. This causes the sign to be printed immediately before the first digit rather than in a fixed location. Each sign after the first also represents one digit.

Example:

Value	Format Field	Print Out
3.75	++++.##	bb+3.75
3.75	----.##	bbb3.75
-3.75	----.##	bb-3.75

A trailing sign can appear only after a decimal point. It terminates the format and prints the appropriate sign (or blank).

Example:

Value	Format Field	Print Out
43.17	***.##+	*43.17+
43.17	&&&.&&-	043.17b
-43.17	###.##+	b43.17-

DOLLAR SIGN (\$)

A dollar sign can be either fixed or floating, and indicates that a \$ is to be printed.

A fixed dollar sign must be either the first or second character in the format field. If it is the second character then + or - must be the first.

Example:

Value	Format Field	Print Out
34.2	\$##.##	\$34.20
34.2	+\$##.##	+\$34.20
-34.2	+\$##.##	-\$ 34.20

Floating dollar signs must start as either the first or second character in the format field and continue to the decimal point. If the floating dollar signs start as the second character then + or - must be the first. Each dollar sign after the first also represents one digit.

Example:

Value	Format Field	Print Out
34.2	\$\$\$\$.##	bb\$34.20
34.2	+\$\$\$\$\$.##	+bb\$34.20
1572563.41	\$\$,\$\$\$,\$\$\$,##+	\$1,572,563.41+

NOTE: There can only be one floating character per format field.

NOTE: +, - or \$ in other than proper positions will give strange results.

String Formats:

The format characters for string format fields are:

- % - Indicates the string is to be right justified.
- ! - indicates the string is to be left justified.

If there are more characters in the string than in the format field, than the string is truncated.

Example:

Value	Format Field	Print Out
ABC	%%%	bABC
ABC	!!!!	ABCb
ABC	%%	AB
ABC	!!	AB

ESCAPE CHARACTER (/)

The escape character (/) does not terminate the format field but will cause the next character to be printed, thus allowing the user to insert a character in the middle of the printing of a number.

Example: PRINT USING "###/-####",2551472
 prints 255-1472

Example: 100 AREA = 408
 200 NUM = 2551472
 300 PHONE = (AREA*1E+7)+NUM
 400 DIM F\$(20)
 500 F\$ = "({##/})###/-####"
 600 PRINT USING F\$,PHONE
 700 END

 the result: (408)255-1472

NOTE: Improperly specified format fields can give some very strange results.

NOTE: The function of "," and ";" in PRINT are overridden in the expression list of PRINT USING, but when file number "ch" is given then the following "," or ";" have the same meaning as in PRINT. So to avoid an initial tabbing, use a semicolon (;).

Example: PRINT #5; USING A\$,B

Will print B in the format specified by A\$ to the file or device associated with file number 5.

Example: PRINT USING "## /* #=###",12,5,5*12

 12 * 5=60

Example: PRINT USING "TOTAL=##.#+",72.68

 TOTAL=72.7+

Example: 100 DIM A\$(10) : A\$="TOTAL="
 200 DIM F\$(10) : F\$="!!!!!!##.#+"
 300 PRINT USING F\$,A\$,72.68

 TOTAL=72.7+

NOTE: IF there are more expressions in the expression list than there are format fields, the format fields will be reused.

Example: PRINT USING "XX##",25,19,7

 will print XX25XX19XXb7

WARNING: A format string must contain at least one format field. If the format string contains only non-format characters, those characters will be printed repeatedly in the search for a format field.

5.20 PROTECT

Format: PROTECT filespec

Examples: PROTECT D:*.COM
100 PROTECT "D2:JUNK.BXL"

The PROTECT allows you to protect your programs stored on disk from being erased or overwritten. This command is very similar to the OS/A+ and DOS XL PROTECT command, except that there are no default file specifications.

5.21 PUT (PU.)

Format: PUT #chan,aexp

Examples: 100 PUT #6,ASC("A")
200 PUT #0,4*13

PUT is the opposite of GET in that it outputs a single byte of information whereas GET inputs a single byte of information. The data output is aexp, and it is put to the device specified by chan.

NOTE: for a program example using PUT, see section 5.31

5.22 READ

Format: READ var [,var...]

Examples: 100 READ A,B,C,D,E
110 DATA 12,13,14,15,16

100 READ A\$,B\$,C\$,D\$,E\$
110 DATA EMBEE, EVELYN, CARLA

The READ command is always used in conjunction with the DATA command. Its function is simply to read the next piece of data out of the DATA list and put it into one of the variables specified. If a group of variables are used, then the first piece of available data (see RESTORE, 4.12) is put into the first variable given, the second piece of data into the second variable given, and so on.

The type of the variable in the READ statement (svar or avar) must correspond to the type of the data which is being read.

If the second example above was executed as a program with no additional lines, an error would result since there are fewer data items than variables to be READ.

The following program totals a list of numbers in a DATA statement:

```
10 FOR N=1 TO 5
20 READ D
30 M=M+D
40 NEXT N
50 PRINT "SUM TOTAL EQUALS ";M
60 END
70 DATA 30,15,106,87,17
```

The program, when executed, will print the statement:

SUM TOTAL EQUALS 255.

NOTE: a Direct mode READ will only read data if a DATA statement exists in the program or on the line following the READ.

5.23 RENAME

Format: RENAME "filespec,filename"

Example: RENAME "D2:NEW.DAT,OLD.BAK"

RENAME allows you to rename file(s) from BASIC XL. Note that the comma shown MUST be imbedded in the string used as the file parameter.

CAUTION: It is strongly suggested that wild cards (* and ?) NOT be used when RENAMEing. Also, the second filename may NOT include the disk specifier (Dn:).

5.24 RGET

Format: RGET #ch, | svar [,svar...] |
 | avar [,avar...] |

Example: (see below)

RGET allows the user to retrieve fixed length records from the device or file associated with file number "ch" and assign the values to string or numeric variables.

NOTE: The type of the element in the file must match the type of the variable (ie. they must both be strings or both be numeric).

Example: 1) 100 RPUT #3,C
 :
 2) 200 RGET #1,A\$

If 1) is a statement in a program used to generate a file and 2) is a statement in another program used to read the same file, an error will result, since 'C' is a numeric variable and 'A\$' is a string variable.

NOTE: When the type of element is string, then the DIMensioned length of the element in the file must be equal to the DIMensioned length of the string variable.

Example: 1) 100 DIM A\$(100)
 :
 800 RPUT #3,A\$

 2) 100 DIM X\$(200)
 :
 800 RGET #2,X\$

If 1) is a section of a program used to write a file and 2) is a section of another program used to read the same file, then an error will occur as a result of the difference in DIM values.

NOTE: RGET sets the correct length for a string variable (the length of a string variable becomes the actual length of the string that was RPUT - not necessarily the DIM length).

Example: 1) 100 DIM A\$(10)
 200 A\$ = "ABCDE"
 :
 800 RPUT #4,A\$

 2) 100 DIM X\$(10)
 200 X\$ = "HI"
 :
 800 RGET #6,X\$
 900 PRINT LEN(X\$),X\$

If 1) is a section of a program used to create a file and 2) is a section of another program used to read the file then it will print:

5 ABCDE

5.25 RPUT

Format: RPUT #ch, exp [,exp...]

Example: (see below)

RPUT allows the user to output fixed length records to the device or file associated with "ch". Each "exp" creates an element in the record.

NOTE: A numeric element consists of one byte which indicates a numeric type element and 6 bytes of numeric data in floating point format.

A string element consists of one byte which indicates a string type element 2 bytes of string length, 2 bytes of DIMensioned length, and then X bytes where X is the DIMensioned length of the string.

Example: 100 DIM A\$(6)
 200 A\$ = "XY"
 300 RPUT #3,B,A\$,10

puts 3 elements to the device or file associated with file number 3. The first element is numeric (the value of B). The second element is a string (A\$) and the third is a numeric (10). The record will be 26 bytes long, (7 bytes for each numeric, 5 bytes for the string header and 6 bytes (the DIM length) of string data).

5.26 SAVE (S.)

Format: SAVE filespec

Example: SAVE D1:YVONNE.PAT
 100 SAVE "C:"

The SAVE command allows you to save the tokenized form of a BASIC XL program to any device. A file saved using this command may then be read back into program memory using the LOAD command or loaded and automatically executed using the RUN command.

5.27 STATUS (ST.)

Format: STATUS #chan,avar

Example: 350 STATUS #1,Z

The STATUS command calls the STATUS routine for the specified device (chan). The status of the STATUS command (see ERROR MESSAGES, Appendix B) is stored in the specified variable (avar). This may be useful for devices such as the RS-232 interface.

5.28 TAB

Format: TAB [#ch,] aexp

Example: TAB #2,20

TAB outputs spaces to the device or file specified by ch (or the screen) up to column number "aexp". The first column is column 0.

NOTE: The column count is kept for each device and is reset to zero each time a carriage return is output to that device. The count is kept in AUX2 of the IOCB. (See OS documentation).

NOTE: If "aexp" is less than the current column count, a carriage return is output and then spaces are put out up to column "aexp".

5.29 UNPROTECT (UNP.)

```
Format:      UNPROTECT filespec

Examples:    100 UNPROTECT "D2:JUNK.BAS
              UNP. D:JUNK
```

The UNPROTECT command allows you to unprotect disk files which have been protected using the PROTECT command. This command is very similar to the OS/A+ and DOS XL command UNProtect, but there are no default file specifications in the BASIC XL version.

5.30 XIO (X.)

```
Format:      XIO cmdno, #chan,aexpl,aexp2,"filespec

Example:     XIO 18,#6,0,0,"S:"
```

The XIO command is a general input/output statement used for special operations. The parameters for this command are defined as follows:

cmdno Number for stands for the particular command to be performed.

cmdno	operation	example
----	-----	-----
3	OPEN	Same as BASIC OPEN
5	GET RECORD	These 4 commands are
7	GET CHARACTERS	similar to BASIC INPUT,
9	PUT RECORD	GET, PRINT, and PUT,
11	PUT CHARACTERS	respectively.
12	CLOSE	Same as BASIC CLOSE
13	STATUS REQUEST	Same as BASIC STATUS
17	DRAW LINE	Same as BASIC DRAWTO
18	FILL	See Section 9
32	RENAME	XIO 32,#1,0,0,"D:TEMP,CAROL"
33	DELETE	XIO 33,#1,0,0,"D:TEMP.BAS"
35	LOCK FILE	XIO 35,#1,0,0,"D:TEMP.BAS"
36	UNLOCK FILE	XIO 36,#1,0,0,"D:TEMP.BAS"
37	POINT	Same as BASIC POINT
38	NOTE	Same as BASIC NOTE
254	FORMAT	XIO 254,#1,0,0,"D2:"

chan Device number (same as in OPEN). Most of the time it is ignored, but must be preceded by #.

aexpl
aexp2 Two auxiliary control bytes. Their usage depends on the particular device and command. In most cases, they are unused and are set to 0.

filespec String expression that specifies the device. Must be enclosed in quotation marks. Although some commands do not look at the filespec, it must still be included in the statement.

NOTE: It is highly recommended that the BASIC XL user avoid XIO cmdno's 3,5,7,9,11,12,17,37 and 38. BASIC XL users should find all these, as well as cmdno's 32 thru 36, totally unnecessary.

5.31 An Example Program

The following subroutine reads in a binary file using OPEN, GET, BGET, CLOSE, and PRINT.

NOTE: lines 1020 through 1030 test the file to see if it is segmented, so you can load in multi-segment files with this subroutine.

```

1000 TRAP 1090
1010 OPEN #1,4,0,"D:FILE.OBJ"
1020 GET #1,L : GET #1,H
1030 IF L=$FF AND H=$FF THEN GET #1,L : GET #1,H
1040 START = H*256 + L
1050 GET #1,L : GET #1,H
1060 FINISH = H*256+L
1070 BGET #1, START, FINISH - START + 1
1080 GOTO 1020
1090 IF ERR(0)=136 THEN CLOSE #1 : RETURN
1100 PRINT "UNEXPECTED ERROR #";ERR(0);" AT LINE "; ERR(1)
1110 STOP

```

A function performs a computation and returns the result (usually a number) for either a print-out or additional computational use. Each function described in this chapter may be used in either Direct or Deferred mode.

This chapter describes the following functions:

Arithmetic Functions

ABS	INT	RND
CLOG	LOG	SGN
EXP	RANDOM	SQR

Trigonometric Functions

ATN	RAD
COS	SIN
DEG	

String Functions

ASC	LEFT\$	RIGHT\$
CHR\$	LEN	STR\$
FIND	MID\$	VAL

Game Controller Functions

HSTICK	PTRIG	VSTICK
PADDLE	STICK	
PEN	STRIG	

Player/Missile Functions

BUMP	PMADR
------	-------

Special Purpose Functions

ADR	ERR	PEEK	TAB
DPEEK	FRE	POKE	USR
DPOKE	HEX\$	SYS	

6.1 Arithmetic Functions

6.1.1 ABS

Format: ABS(aexp)

Example: 100 AB = ABS(-190)

Returns the absolute value of a number without regard to whether it is positive or negative. The returned value is always positive.

6.1.2 CLOG

Format: CLOG (aexp)

Example: 100 C = CLOG(83)

Returns the logarithm to the base 10 of the variable or expression in parentheses. CLOG(0) gives an error, and CLOG(1) is 0.

6.1.3 EXP

Format: EXP(aexp)

Example: 100 PRINT EXP(3)

Returns the value of e (approximately 2.71828283), raised to the power specified by the expression in parentheses. In the example given above, the number returned is 20.0855365.

6.1.4 INT

Format: INT(aexp)

Example: 100 I = INT(3.445) : REM I now = 3
100 X = INT(-14.66778) : REM X now = -15

Returns the greatest integer less than or equal to the value of the expression. This is true whether the expression evaluates to a positive or negative number. Thus, in our first example above, I is used to store the number 3. In the second example, X is used to store the number -15 (the first whole number that is less than or equal to -14.66778). This INT function should not be confused with the function used on calculators that simply truncates all decimal places.

6.1.5 LOG

Format: LOG(aexp)

Example: 100 L = LOG(67.89/2.57)

Returns the natural logarithm of the number or expression in parentheses. LOG(0) gives an error, and LOG(1) is 0.

6.1.6 RANDOM

Format: RANDOM(aexp1[,aexp2])

Example: 10 X = RANDOM(99)
10 Y = RANDOM(20,30)

The RANDOM function allows you access to a random number generator which does more than return a number between 0 and 1, as RND does. When used with one aexp (as in the first example), the value returned will be between 0 and the aexp value, inclusive. When used with two aexps (as in the second example), the value returned will be between the value of the first aexp and the value of the second aexp, inclusive.

6.1.7 RND

Format: RND(aexp)

Example: 10 A = RND(0)

Returns a hardware-generated random number between 0 and 1, but never returns 1. The variable or expression in parentheses following RND is a dummy and has no effect on the numbers returned. However, the dummy expression must be included.

6.1.8 SGN

Format: SGN(aexp)

Example: 100 X = SGN(-199) : REM -1 is returned

Returns a -1 if aexp evaluates to a negative number; a 0 if aexp evaluates to 0, or a 1 if aexp evaluates to a positive number.

6.1.9 SQR

Format: SQR(aexp)

Example: 100 PRINT SQR(100) REM 10 is printed

Returns the square root of the aexp which must be positive.

6.1.10 An Example Program

The following program prints out some information on an INPUTted number, using the arithmetic functions ABS, INT, SQR, CLOG, LOG, and EXP.

```
100 GRAPHICS 1 : REM set up screen
110 PRINT "Number to Manipulate> ";
120 INPUT #0, X : REM get the number
130 PRINT #6; ASC$(125) : REM clear screen
140 PRINT #6; "ABS.: "; ABS(X) : REM absolute value
150 PRINT #6
160 PRINT #6; "INT.: "; INT(X) : REM integer value
170 PRINT #6
180 PRINT #6; "SQRT: "; SQR(ABS(X)) : REM square root
190 PRINT #6
200 PRINT #6; "CLOG: "; CLOG(ABS(X)) : REM common log
210 PRINT #6
220 PRINT #6; "NLOG: "; LOG(ABS(X)) : REM natural log (ln)
230 PRINT #6
240 PRINT #6; "EXP.: "; EXP(X) : REM exponential (e^X)
250 GOTO 110
```

6.2 Trigonometric Functions

6.2.1 ATN

Format: ATN(aexp)

Example: 100 X = ATN(1.0)

Returns the arctangent of the variable or expression in parentheses. If in DEG mode (see section 6.2.3), the returned value is given in degrees, otherwise it is given in radians.

6.2.2 COS

Format: COS(aexp)

Example: 100 C = COS(X+Y+Z)

Returns the trigonometric cosine of the expression in parentheses. The expression is evaluated as an angle in radian terms unless the DEG command has been used.

6.2.3 DEG and RAD

Format: DEG
RAD

Example: 100 DEG
100 RAD

These two statements allow the programmer to specify degrees or radians for trigonometric function computations. The computer defaults to radians unless DEG is specified. Once the DEG statement has been executed, RAD must be used to return to radians.

See Appendix E for the additional trigonometric functions that can be derived.

6.2.4 SIN

Format: SIN(aexp)

Example: 100 X = SIN(Y)

This function returns the trigonometric sine of aexp. The expression is evaluated as an angle in radian terms unless the DEG command has been used.

6.2.5 An Example Program

The following program demonstrates the use of DEG, COS, and SIN by plotting three concentric circles on the screen.

```
10 GRAPHICS 7 : REM set up screen
20 DEG : REM degree mode for trig functions
30   FOR J=1 TO 3 : REM 3 circles
40     COLOR J : REM each circle a different color
50     FOR I=1 TO 360 : REM plot each point in a full circle
60       PLOT 80+INT(J*10*COS(I)), 40+INT(J*10*SIN(I))
70     NEXT I
80   NEXT J
```

6.3 String Functions

6.3.1 ASC

Format: ASC(sexp)

Examples: 100 A = ASC(A\$)

This function returns the ATASCII code number for the first character of the string expression (sexp). This function can be used in either Direct or Deferred mode.

If A\$= "ABC", then
 ASC(A\$) produces 65
 ASC(A\$(2)) produces 66

6.3.2 CHR\$

Format: CHR\$(aexp)

Examples: 100 PRINT CHR\$(65)
 100 A\$ = CHR\$(65)

This character string function returns the character, in string format, represented by the ATASCII code number in parentheses. Only one character is returned. In the above examples, the letter A is returned. Using the ASC and CHR\$ functions, the following program prints the upper case and lower case letters of the alphabet:

```
10 FOR I=0 TO 25
20 PRINT CHR$(ASC("A")+I);CHR$(ASC("a")+I)
30 NEXT I
```

NOTE: There can be only one STR\$ and only one CHR\$ in a

logical comparison. (This is because BASIC XL uses a buffer in a fixed location to create the temporary string which both of these functions produce, and there is only one such buffer.)

6.3.3 FIND

Format: FIND(sexp1,sexp2,aexp)

Example: PRINT FIND ("ABCDXXXXABC","BC",N)

FIND is an efficient, speedy way of determining whether any given substring is contained in any given master string.

FIND will search sexp1, starting at position aexp, for sexp2. If sexp2 is found, the function returns the position where it was found, relative to the beginning of sexp1. If sexp2 is not found, a 0 is returned.

In the example above, the following values would be PRINTed:

```
2 if N=0 or N=1
9 if N>2 and N<10
0 if N>=10
```

More Examples:

```
1) 10 DIM A$(1)
    20 PRINT "INPUT A SINGLE LETTER:
    30 PRINT "Change/Erase/List"
    40 INPUT "CHOICE ?",A$
    50 ON FIND("CEL",A$,0) GOTO 100,200,300
```

An easy way to have a vector from a menu choice:

```
2) 100 DIM A$(10): A$="ABCDEFGHIJ"
    110 PRINT FIND (A$,"E",3)
    120 PRINT FIND (A$(3),"E",0)
```

Line 110 will print "5" while 120 will print "3". Remember, the position returned is relative to the start of the specified string.

```
3) 100 INPUT "20 CHARACTERS, PLEASE:",A$
    110 ST=0
    120 F=FIND(A$,"A",ST):IF F=0 THEN STOP
    130 IF A$(F+1,F+1)<>"B" AND A$(F+1,F+1)<>"C"
        THEN ST=F+1:GOTO 120
    140 PRINT "FOUND 'AB' OR 'AC'"
```

This illustrates the importance of the aexp's use as a starting position.

6.3.4 LEFT\$

Format: LEFT\$(svar,aexp)

Example: 100 A\$=LEFT\$("ABCDE",3)
200 PRINT LEFT\$("ABCD",5)

The LEFT\$ function returns the leftmost 'aexp' characters of the string 'svar'. If aexp is greater than the number of characters in svar, no error occurs and the entire string svar is returned.

In the first example, A\$ is equated to "ABC"x, and in the second example, the entire string "ABCD" is printed.

6.3.5 LEN

Format: LEN(sexp)

Example: 100 PRINT LEN(A\$)

This function returns the length in bytes of the designated string. This information may then be printed or used later in a program. The length of a string variable is simply the index for the character which is currently at the end of the string. Strings have a length of 0 until characters have been stored in them. It is possible to store into the middle of the string by using subscripting. However, the beginning of the string will contain garbage.

The following routine illustrates one use of the LEN function:

```
10 A$="ATARI"          10 DIM AR$(3,0)
20 PRINT LEN(A$)        20 AR$(2;)= "ATARI"
                        30 PRINT LEN(AR$(2;))
```

The result of running either of the above programs would be 5.

6.3.6 MID\$

Format: MID\$(svar,aexpl,aexp2)

Example: A\$=MID\$("ABCDEFGH",2,4)

MID\$ allows you to get a substring from the middle of another string. The substring starts at the 'aexpl'th character of svar, and is 'aexp2' characters long.

If aexp1 equals 0 an error occurs (since there is no zeroeth character of a string), but if aexp1 is greater than the length of svar no error occurs (and no characters are returned).

aexp2 is allowed any positive number (including 0), but if its value makes the substring go beyond the length of svar, then the substring returned ends at the end of svar.

In the above example, A\$ is equated to "BCDE".

6.3.7 RIGHT\$

Format: RIGHT\$(svar,aexp)

Example: A\$=RIGHT\$("123456",4)

This function is used to return the rightmost 'aexp' characters of 'svar'. If aexp is greater than the number of characters in svar, then the entire string 'svar' is returned.

In the above example, A\$ is equated to "3456".

6.3.8 STR\$

Format: STR(aexp)

Example: A\$=STR\$(65)

This function returns the string form of the number in parentheses. The above example would return the actual number 65, but it would be recognized by the computer as a string.

NOTE: There can only be one STR\$ and only one CHR\$ in a logical comparison. For example, A=STR\$(1)>STR\$(2) is not valid and will not work correctly.

6.3.9 VAL

Format: VAL(sexp)

Example: 100 A=VAL(A\$)

This function is the opposite of the STR\$ function, in that it returns the number represented by a string, providing that the string is indeed a string representation of a number. Using this function, the

computer can perform arithmetic operations on strings as shown in the following example program:

```
10 DIM B$(5)
20 B$="10000"
30 B=SQR(VAL(B$))
40 PRINT "THE SQUARE ROOT OF ";B$;" IS ";B
```

Upon execution, the screen displays:

THE SQUARE ROOT OF 10000 IS 100.

It is not possible to use the VAL function with a string that does not start with a number, or that cannot be interpreted by the computer as a number. It can, however, interpret floating point numbers (e.g., VAL("1E9") would return the number 1000000000).

6.3.10 An Example Program

The following program inputs a three word string, cuts it up into the separate words through the use of LEFT\$, MID\$, and RIGHT\$, and then prints out the ATASCII value of each letter in each word using ASC. Note that this program also uses the LEN and FIND functions.

```
100 PRINT "Give me a three word string with each"
110 INPUT "word separated by a space> ",S$
120 POS1=FIND(S$," ",0) : REM find end of 1st word
130 L$=LEFT$(S$,POS1-1) : REM fill 1st word string
140 POS2=FIND(S$," ", POS1) : REM find 2nd word
150 M$=MID$(S$,POS1+1,POS2-POS1-1) : REM fill 2nd word string
160 R$=RIGHT$(S$,LEN(S$)-POS2) : REM fill 3rd word string
170 PRINT "**** ";L$ : REM print 1st word
180 FOR I=1 TO LEN(L$) : REM print ASC value of each letter
190 PRINT ,L$(I,I); " : "; ASC(L$(I))
200 NEXT I
210 PRINT "**** ";M$ : REM print 2nd word
220 FOR I=1 TO LEN(M$) : REM print ASC value of each letter
230 PRINT ,M$(I,I); " : "; ASC(M$(I))
240 NEXT I
250 PRINT "**** ";R$ : REM print 3rd word
260 FOR I=1 TO LEN(R$) : REM print ASC value of each letter
270 PRINT ,R$(I,I); " : "; ASC(R$(I))
280 NEXT I
290 GOTO 100
```

NOTE: lines 130, 150, and 160 could have been coded as follows:

```
130 L$=S$(1,POS1-1)
150 M$=S$(POS1+1,POS2-1)
160 R$=S$(POS2+1)
```

6.4 Game Controller Functions

6.4.1 HSTICK

Format: HSTICK(aexp)

Example: 100 IF HSTICK(0)>0 THEN PRINT "MOVE RIGHT"

The HSTICK function returns an easily usable code for horizontal movement of a given joystick. aexp is simply the number of the joystick port (0 - 3), and the values returned (and their meanings) are as follows:

- +1 if the joystick is pushed right
- 1 if the joystick is pushed left
- 0 if the joystick is horizontally centered

6.4.2 PADDLE

Format: PADDLE(aexp)

Example: PRINT PADDLE(3)

This function returns the current value of a particular paddle. aexp is the number of the paddle port (0 - 7). The value returned will be between 1 and 228, with the number increasing as the knob is turned counterclockwise.

6.4.3 PEN

Format: PEN(aexp)

Example: PRINT "light pen at X=";PEN(0)

The PEN function simply reads the ATARI light pen registers and returns their contents to the user. The number specified by aexp is interpreted as follows:

- PEN(0) reads the horizontal position register
- PEN(1) reads the vertical position register

6.4.4 PTRIG

Format: PTRIG(aexp)

Example: 100 IF PTRIG(1)=0 THEN PRINT "MISSILES FIRED!"

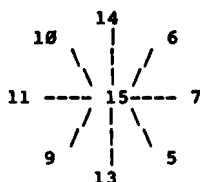
The PTRIG function returns a status of 0 if the trigger button of the designated paddle is pressed. Otherwise, it returns a value of 1. The aexp must be a number between 0 and 7 as it designates the paddle.

6.4.5 STICK

Format: STICK(aexp)

Example: 100 PRINT STICK(3)

This function works exactly the same way as the PADDLE command, but is used with the joystick controllers. aexp is the number of the joystick port (0 - 3). The following diagram shows the values returned by this function:



COMMENT: this function was the only means given to access the joystick with original Atari BASIC. For most purposes, HSTICK and VSTICK are much easier to use and to work with.

6.4.6 STRIG

Format: STRIG(aexp)

Example: 100 IF STRIG(1)=0 THEN PRINT "FIRE TORPEDO"

The STRIG function works the same way as the PTRIG function, except that it is used with the joysticks instead of the paddles.

6.4.7 VSTICK

Format: VSTICK(aexp)

Example: IF VSTICK(0)<0 THEN PRINT "MOVE DOWN"

The VSTICK function returns an easily usable code for vertical movement of a given joystick. aexp is simply the number of the joystick port (0 - 3), and the values returned (and their meanings) are as follows:

- +1 if the joystick is pushed up
- 1 if the joystick is pushed down
- 0 if the joystick is vertically centered

6.4.8 An Example Program

The following program creates a simple GRAPHICS mode 5 sketchpad using the game controller functions HSTICK, VSTICK, and STRIG to move and draw.

```
100 GRAPHICS 5 : REM set up screen
110 COL=40 : REM middle of screen
120 ROW=20
130 COLOR 2 : REM drawing a cursor color
140 PLOT COL, ROW : REM plot cursor
150 FOR I=1 TO 15 : NEXT I : REM delay loop
160 IF STRIG(0)=1 THEN COLOR 0 : PLOT COL, ROW : REM dont draw point
170 COL=COL+HSTICK(0) : REM check for movement
180 ROW=ROW-VSTICK(0)
190 IF COL<0 THEN COL=0 : REM screen bounds checking
200 IF COL>79 THEN COL=79
210 IF ROW<0 THEN ROW=0
220 IF ROW>39 THEN ROW=39
230 FOR I=1 TO 25 : NEXT I : REM delay loop
240 GOTO 130 : REM repeat
```

6.5 Player/Missile Functions

For examples showing the use of the P/M functions, see section 8.13.

6.5.1 BUMP

Format: BUMP(pmnum,aexp)

Example: IF BUMP(4,1) THEN B=BUMP(0,8)

BUMP accesses the collision registers of the Atari and returns a 1 (collision occurred) or 0 (no collision occurred) as appropriate for the pair of objects specified. Note that the second parameter (the aexp) may be either a player number or playfield number (see section 8.2 for the appropriate number).

Valid BUMPs: PLAYER to PLAYER (0-3 to 0-3)
 MISSILE to PLAYER (4-7 to 0-3)
 PLAYER to PLAYFIELD (0-3 to 8-11)
 MISSILE to PLAYFIELD (4-7 to 8-11)

NOTE: BUMP (p,p), where the p's are 0 through 3 and identical, always returns 0.

NOTE: It is advisable to reset the collision registers if you have not checked them in a long time or after you are through checking them at any given point in a

program. You can do this by using the following statement:

POKE 53278,0

6.5.2 PMADR

Format: PMADR(aexp)

Example: P0=PMADR(0)

This function may be used in any arithmetic expression and is used to obtain the memory address of any player or missile. It is useful when you wish to MOVE, POKE, BGET, etc. data to (or from) a player area. (See section 8.13 for examples of its use, and section 8.2 for a description of the aexp values.)

NOTE: PMADR(m) -- where m is a missile number (4 through 7) returns the same address for all missiles.

6.6 Special Purpose Functions

6.6.1 ADR

Format: ADR(svar)

Example: ADR(A\$)
ADR(B\$(5;))

Returns the decimal memory address of the string specified by the expression in parentheses. Knowing the address enables the programmer to pass the information to USR routines, etc. (See USR and Appendix D).

6.6.2 DPEEK

Format: DPEEK(aexp)

Example: PRINT "variable table is at ";DPEEK(130)

The DPEEK function is very similar to the PEEK function, except that it allows you to look two consecutive bytes of information. This is especially useful when looking at two byte locations containing address information, as in the above example. If you did this example using PEEKs, it would look like:

PRINT "variable name table is at ";
PRINT PEEK(130)+(PEEK(131)*256)

It is easy to see that using DPEEK is much easier.

6.6.3 DPOKE

Format: DPOKE aexp1,aexp2

Example: DPOKE 88,32768

DPOKE is similar to POKE, except that it allows you to put two bytes of data into memory instead of one. aexp1 is the address where you want the data to go, and aexp2 is the data itself. In the above example, the address of the upper left-hand corner of the screen (this address is stored at locations 88 and 89) is changed to 32768. To do this using POKEs, you would need to do an amazing amount of math to get the right number into each of the two bytes.

6.6.4 ERR

Format: ERR(aexp)

Example: PRINT "ERROR ";ERR(0); " OCCURRED AT LINE ";ERR(1)

This function -- in conjunction with TRAP, CONT, and GOTO allows the BASIC XL programmer to effectively diagnose and dispatch virtually any run-time error.

ERR(0) returns the last run-time error number
ERR(1) returns the line number where the error occurred

Example:

```
100 TRAP 200
110 INPUT "A NUMBER, PLEASE >>",NUM
120 PRINT "A VALID NUMBER" : END
200 IF ERR(0)=8 THEN GOTO ERR(1)
210 PRINT "UNEXPECTED ERROR #";ERR(0)
```

6.6.5 FRE

Format: FRE(aexp)

Example: PRINT FRE(0)
100 IF FRE(0)<1000 THEN
PRINT "MEMORY CRITICAL"

This function returns the number of bytes of user RAM left. Its primary use is in Direct mode with a dummy variable (0) to inform the programmer how much memory space remains for completion of a program. Of course FRE can also be used within a BASIC program in Deferred mode.

6.6.6 HEX\$

Format: HEX\$(aexp)

Example: 100 PRINT HEX\$(X+7)
 200 A\$=HEX\$(83)
 210 PRINT "\$";A\$(3,4)

This function will convert aexp to a four digit hexadecimal number.

The second example shows how you can obtain a two digit hex number for printing or other manipulation.

NOTE: no "\$" is placed in front of the number.

6.6.7 PEEK

Format: PEEK(aexp)

Example: 1000 IF PEEK (4000) = 255 THEN PRINT "255"
 100 PRINT "LEFT MARGIN IS";PEEK(82)

Returns the contents of a specified memory address location (aexp). The address specified must be an integer or an arithmetic expression that evaluates to an integer between 0 and 65535 and represents the memory address in decimal notation (not hexadecimal). The number returned will also be a decimal integer with a range from 0 to 255. This function allows the user to examine either RAM or ROM locations. In the first example above, the PEEK is used to determine whether location 4000 (decimal) contains the number 255. In the second example, the PEEK function is used to examine the left margin.

6.6.8 POKE

Format: POKE aexp1,aexp2

Example: POKE 82,10
 100 POKE 82,20

Although this is not a function, it is included in this section because it is closely associated with the PEEK function. This POKE command inserts data into the memory location or modifies data already stored there. In the above format, aexp1 is the decimal address of the location to be poked and aexp2 is the data to be poked. Note that this number is a decimal number between 0 and 255. POKE cannot be used to alter ROM locations. In gaining familiarity with this command it is advisable to look at the memory location with a PEEK

and write down the contents of the location. Then, if the POKE doesn't work as anticipated, the original contents can be poked back into the location.

The above Direct mode example changes the left screen margin from its default position of 2 to a new position of 10. In other words, the new margin will be 8 spaces to the right. To restore the margin to its normal default position, press <SYSTEM RESET>.

6.6.9 SYS

Format: SYS(aexp)

Example: 100 IF SYS(0)=0 THEN SET 0, 128

The SYS function is used to find out the status of a given BASIC XL system function. These system functions can be changed using the SET command, and SYS allows you to find out what any current value is. aexp is the number of the system function as defined in the SET section (3.15).

6.6.10 TAB

Format: TAB(aexp)

Example: PRINT #3,"columns:";TAB(20);20;TAB(30);30

The TAB function's effect is identical with that of the TAB statement (section 5.28). The difference is that, for PRINT USING statements, an imbedded TAB function simplifies the programmers task greatly.

TAB will output ATASCII space characters to the current PRINT file or device (#3 in our example). Sufficient spaces will be output so that the next item will print in the column specified (only if TAB is followed by a semi-colon, though). If the column specified is less than the current column, a RETURN will be output first.

CAUTION: The TAB function will output spaces on some device whenever it is used; therefore, it should be used ONLY in PRINT statements.

6.6.11USR

Format: USR(aexp1 [,aexp2][aexp3...])

Example: 100 RESULT = USR (ADD1,A*2)

This function returns the results of a machine-language subroutine. The first expression, aexp1, must be an

integer or arithmetic expression that evaluates to an integer that represents the decimal memory address of the machine language routine to be performed. The input arguments aexp2,aexp3,etc., are optional. These should be arithmetic expressions within a decimal range of 0 through 65535. A non-integer value may be used; however, it will be rounded to the nearest integer.

These values will be converted from BASIC's Binary Coded Decimal (BCD) floating point number format to a two-byte binary number, then pushed onto the hardware stack.

The arguments are pushed in the reverse of the order given, so the assembly language program may then pull them in proper forward order. Additionally, the one-byte count of parameters is pushed onto the stack and MUST be popped by the USR routine (except see section 3.15, the SET command).

Also, if all arguments are properly pulled from the stack, then the USR routine may return to BASIC XL by simply executing an RTS instruction. And, finally, the routine may return a single 16-bit value to BASIC XL (as the "value" of the USR function) by placing a result in FR0 and FR0+1 (\$D4 and \$D5) before returning.

Example: the following example uses a USR call to XOR two numbers (the arguments to the USR routine) and then return that value to BASIC XL.

BASIC XL statement:

```
-----
      PRINT HEX$(USR($680,$3FFA,$2972))
```

USR routine at \$680:

```
-----
FR0  =   $D4
      *=   $680
      PLA      ; get number of arguments
      CMP #2   ; see if it's 2
      BNE *    ; loop forever if wrong num. of args.
      PLA      ; get high byte of arg #1
      STA FR0+1 ; store high byte
      PLA      ; get low byte of arg #1
      STA FR0   ; store low byte
      PLA      ; get high byte of arg #2
      EOR FR0+1 ; XOR it with high byte of arg #1
      STA FR0+1 ; store result of XOR
      PLA      ; get low byte of arg #2
      EOR FR0   ; XOR it with low byte of arg #1
      STA FR0   ; store result of XOR
      RTS      ; end of USR routine
```

6.6.12 An Example Program

The following program uses the system timer located at \$12, \$13, and \$14 to create a countdown clock. This is done by poking 0 into the low byte of the timer and waiting until it is greater than or equal to 60.

```
100 GRAPHICS 2
110 PRINT #6; CHR$(125) : REM Clear Mode 2 area
120 PRINT : PRINT : PRINT
130 PRINT "COUNTDOWN TIME? ";
140 INPUT #0,X
150 POKE $14,0 : REM set clock = 0
160 PRINT #6; "TIME - ";
170 WHILE X>0 : REM start the countdown
180   POSITION 7,1 : REM get ready to print the new time
190   PRINT #6; USING "##",X; : REM print time left
200   WHILE PEEK($14)<=60 : REM wait until a second has passed
210     ENDWHILE
220   POKE $14,0 : REM reset the clock for the next second
230   X=X-1 : REM decrement number of seconds left
240   ENDWHILE : REM end of countdown loop
250 PRINT CHR$(253) : REM ring the bell
260 GOTO 110 : REM do the whole thing over again
```

This chapter describes the BASIC XL commands used to manipulate the wide variety of screen graphics available on the Atari personal computers. It also describes the BASIC XL command used to manipulate the sound generating mechanism of the Atari computers.

7.1 GRAPHICS (GR.)

Format: GRAPHICS aexp

Example: GRAPHICS 2

This command is used to select one of the nine graphics modes. The table below summarizes the nine modes and the characteristics of each.

The GRAPHICS command automatically opens the graphics area of the screen (S:) on channel #6. As a result of this, it is not necessary to specify a channel number when you want to PRINT to the text window, since it is still open on channel #0.

NOTE: aexp must be positive.

Graphics modes 0, 9, 10, and 11 are full-screen display while modes 1 through 8 are split screen displays. To override the split-screen, add 16 to the mode number (aexp) in the GRAPHICS command. Adding 32 prevents the graphics command from clearing the screen.

To return to graphics mode 0 in Direct mode, press <SYSTEM RESET> or type GR.0 and press <RETURN>.

Gr. Mode	Mode Type	(split)		(full)		Num of
		Cols	Rows	Rows	Colors	
0	TEXT	40	N/A	24	2	
1	TEXT	20	20	24	5	
2	TEXT	20	10	12	5	
3	GRAPHICS	40	20	24	4	
4	GRAPHICS	80	40	48	2	
5	GRAPHICS	80	40	48	4	
6	GRAPHICS	160	80	96	2	
7	GRAPHICS	160	80	96	4	
8	GRAPHICS	320	160	192	1 1/2	
9	GRAPHICS	80	N/A	192	16	
10	GRAPHICS	80	N/A	192	9	
11	GRAPHICS	80	N/A	192	16	

7.1.1 GRAPHICS Mode 0

This mode is the 1-color, 2-luminance(brightness) default mode for the ATARI Personal Computer. It contains a 24 line by 40 character screen matrix. The default margin settings at 2 and 39 allow 38 characters per line. Margins may be changed by poking LMARGN and RMARGN (82 and 83).

Some systems have different margin default settings. The color of the characters is determined by the background color. Only the luminance of the characters can be different. This full-screen display has a blue display area bordered in black (unless the border is specified to be another color). To display characters at a specified location, use one the following method:

```
POSITION aexpl,aexp2 : REM Puts cursor at location
PRINT sexp          : REM specified by aexpl and aexp2.
```

GRAPHICS 0 is also used as a clear screen command either in Direct mode or Deferred mode. It terminates any previously selected graphics mode and returns the screen to the default mode (GRAPHIC 0).

7.1.2 GRAPHICS Modes 1 and 2

These two 5-color modes are Text modes. However, they are both split-screen modes.

Characters printed in Graphics mode 1 are twice the width of those printed in Graphics 0, but are the same height.

Characters printed in Graphics mode 2 are twice the width and height of those in Graphics mode 0.

In the split-screen mode, a PRINT command is used to display characters in either the text window or the graphic window. To print characters in the graphic window, specify channel #6 after the PRINT command.

```
Example: 100 GR. 1
          110 PRINT #6;"A MODE 1 TEST"
```

The default colors depend on the type of character input, as defined in the following table:

Character Type	Color Register	Default Color
Upper case alphabetic	0	Orange
Lower case alphabetic	1	Light Green
Inverse upper case alphabetic	2	Dark Blue
Inverse lower case alphabetic	3	Red
Numbers	0	Orange
Inverse numbers	2	Dark Blue

NOTE: see SETCOLOR to change character colors.

Unless otherwise specified, all characters are displayed in upper case non-inverse form. To print lower case letters and graphics characters, use a POKE 756,226. To return to upper case, use POKE 756,224.

In graphics modes 1 and 2, there is no inverse video, but it is possible to get all the rest of the characters in four different colors (see end of section).

7.1.3 GRAPHICS Modes 3,5, and 7

These three 4-color graphics modes are also split-screen displays in their default state, but may be changed to full screen by adding 16 to the mode number. Modes 3, 5, and 7 are alike except that modes 5 and 7 use more points (pixels) in plotting, drawing, and positioning the cursor; the points are smaller, thereby giving a much higher resolution.

7.1.4 GRAPHICS modes 4 and 6

These two 2-color graphics modes are split-screen displays and can display in only two colors while the other modes can display 4 and 5 colors. The advantage of a two-color mode is that it requires less RAM space. Therefore, it is used when only two colors are needed and RAM is getting crowded. These two modes also have a higher resolution which means smaller points than Graphics mode 3.

7.1.5 GRAPHICS mode 8

This graphics mode gives the highest resolution of all the other modes. As it takes a lot of RAM to obtain this kind of resolution, it can only accomodate a maximum of one color and two different luminances, as mode 0.

7.1.6 GRAPHICS modes 9, 10, and 11

GRAPHICS modes 9, 10, and 11 are the GTIA modes, and are somewhat different from all the other modes. Note that these modes do not allow a text window.

Mode 9 is a one color, 16 luminance mode. The main color is set by the background color, and the luminance values are determined by the information in the screen memory itself. Each pixel is four bits wide, allowing for 16 different values. These values are interpreted as the luminance of the base color for that pixel.

Mode 11 is similar to mode 9 in that the color information is in the screen memory itself, but the information for each pixel is interpreted as a color instead of a luminance. Thus there are 16 colors, all of the same luminance. The luminance is set by the luminance of the background color (default = 6).

Mode 10 is somewhat of a crossbreed of the other two GTIA modes and the normal modes in that it offers lots of colors (like the GTIA modes) and uses the color registers (like the normal modes). However, since mode 10 allows 9 colors, it must use the player color registers as well as the other color registers. Below is a table showing how the pixel values relate to the color registers and what BASIC XL command may be used.

VALUE	REGISTER	REG. ADDRESS	COMMAND
0	PCOLR0	704	PMCOLOR 0
1	PCOLR1	705	PMCOLOR 1
2	PCOLR2	706	PMCOLOR 2
3	PCOLR3	707	PMCOLOR 3
4	COLOR0	708	SETCOLOR 0
5	COLOR1	709	SETCOLOR 1
6	COLOR2	710	SETCOLOR 2
7	COLOR3	711	SETCOLOR 3
8	COLOR4	712	SETCOLOR 4

7.2 COLOR (C.)

Format: COLOR aexp

Examples: 110 COLOR ASC("A")
 110 COLOR 3

The value of the expression in the COLOR statement determines the data to be stored in the display memory for all subsequent PLOT and DRAWTO commands until the next COLOR statement is executed. The value must be positive and is usually an integer from 0 through 255. Non-integers are rounded to the nearest integer. The graphics display hardware interprets this data in different ways in the different graphics modes.

In text modes 0 through 2, the number can be from 0 through 255 (8 bits) and determines the character to be displayed and its color. (The two most significant bits determines the color. This is why only 64 different characters are available in these modes instead of the full 256-character set.)

Graphics modes 3 through 8 are not text modes, so the data stored in the display RAM simply determines the color of each pixel. Two-color or two-luminance modes require either 0 or 1 (1-bit) and four-color modes require 0, 1, 2, or 3. (The expression in the COLOR statement may have a value greater than 3, but only one or two bits will be used.)

The actual color which is displayed depends on the value in the color register which corresponds to the data of 0, 1, 2, or 3 in the particular graphics mode being used. This may be determined by looking in the table at the end of the SETCOLOR section. This table gives COLOR and SETCOLOR relationships for all the GRAPHICS modes.

Note that when BASIC XL is first powered up, the color data is 0, and when a GRAPHICS command (without +32) is executed, all of the pixels are set to 0. Therefore, nothing seems to happen to PLOT and DRAWTO in GRAPHICS 3 through 7 when no COLOR statement has been executed. Correct this by doing a COLOR 1 first.

7.3 DRAWTO (DR.)

Format: DRAWTO aexpl,aexp2

Example: 100 DRAWTO 10,8

This statement causes a line to be drawn from the last point displayed by a PLOT (see PLOT) to the location by aexpl and aexp2. The first expression represents the X coordinate (column) and the second represents the Y-coordinate (row). The color of the line is the same color as the point displayed by the PLOT.

7.4 LOCATE (LOC.)

Format: LOCATE aexpl,aexp2,avar

Example: 150 LOCATE 11,15,X

This command positions the invisible graphics cursor at the specified location in the graphics window, retrieves the data at that pixel, and stores it in the specified arithmetic variable. This gives a number from 0 to 255 for Graphics modes 0 through 2, a 0 or 1 for the 2-color graphics modes, and a 0,1,2, or 3 for the 4-color modes. The two arithmetic expressions specify the X and Y coordinates of the point. LOCATE is equivalent to:

POSITION aexpl,aexp2:GET#6,avar

Doing a PRINT after a LOCATE or GET from the screen may cause the data in the pixel which was examined to be modified. This problem is avoided by repositioning the cursor and putting the data that was read back into the pixel before doing the PRINT. The following program illustrates the use of the LOCATE command:

```
10 GRAPHICS 3+16
20 COLOR 1
30 SETCOLOR 2,10,8
40 PLOT 10,15
50 DRAWTO 15,15
60 LOCATE 12,15,X
70 PRINT X
```

On execution, the program prints the data (1) determined by the COLOR statement which was stored in pixel 12,15.

7.5 PLOT (PL.)

Format: PLOT aexp1,aexp2

Example: 100 PLOT 5,5

The PLOT command is used in graphics modes 3 through 8 to display a point in the graphics window. aexp1 specifies the X-coordinate and aexp2 specifies the Y-coordinate. The color of the plotted point is determined by the hue and luminance in the color register from the last COLOR statement executed. To change this color register, and the color of the plotted point, use SETCOLOR. Points that can be plotted on the screen are dependent on the graphics mode being used. The range of points begins at (0,0), and extends to one less than the total number of rows (X-coordinate) or columns (Y-coordinate).

NOTE: PLOT aexp1,aexp2 is equivalent to:

POSITION aexp1,aexp2 : PUT #6, COLOR

7.6 POSITION (POS.)

Format: POSITION aexp1,aexp2

Example: 100 POSITION 8,12

The POSITION statement is used to place the invisible graphics window cursor at the specified location on the screen (usually precedes a PRINT or PUT statement). This statement can be used in all modes. Note that the cursor does not actually move until an I/O command which involves the screen is issued.

7.7 PUT and GET (as applied to graphics)

Formats: PUT #6,aexp
GET #6,avar

Examples: 100 PUT #6,ASC("A")
200 GET #6,X

In graphics work, PUT is used to output data to the screen display. This statement works hand-in-hand with the POSITION statement. After a PUT (or GET), the cursor is moved to the next location on the screen.

Doing a PUT to device #6 causes the one-byte aexp to be displayed at the cursor position. The byte is either an ATASCII code byte for a particular character (modes 0-2) or the color data (modes 3-8).

GET is used to input the code byte of the character displayed at the cursor position, into the specified arithmetic variable. The values used in PUT and GET correspond to the values in the COLOR statement. (PRINT and INPUT may also be used.)

NOTE: doing a PRINT after a LOCATE or GET from the screen may cause the data in the pixel which was examined to be modified. To avoid this problem, reposition the cursor and put the data that was read back into the pixel before doing the PRINT.

7.8 SETCOLOR (SE.)

Format: SETCOLOR aexp1,aexp2, aexp3

Example: 100 SETCOLOR 0,1,4

This statement is used to choose the particular hue and luminance to be stored in the specified color register. The parameters of the SETCOLOR statement are defined below:

aexp1 = Color register (0-4 depending on graphics mode)
aexp2 = Color hue number (0-15 -- see the table below)
aexp3 = Color luminance (must be an even number between 0 and 14; the higher the number, the brighter the display. 14 is almost pure white.)

SETCOLOR aexp2	Color	SETCOLOR aexp2	Color
-----	-----	-----	-----
0	Gray	8	Blue
1	Gold	9	Light Blue
2	Orange	10	Turquoise
3	Red-Orange	11	Green-Blue
4	Pink	12	Green
5	Purple	13	Yellow-Green
6	Purple-Blue	14	Orange-Green
7	Blue	15	Light Orange

Note: Colors will vary with type and adjustment of TV or monitor used.

The ATARI display hardware contains five color registers, numbered from 0 through 4. The Operating System (OS) has five RAM locations (COLOR0 through COLOR4, see Appendix I - Memory Locations) where it keeps track of the current colors. The SETCOLOR statement is used to change the values in these RAM locations. (The OS transfers these values to the hardware registers every television frame.)

The SETCOLOR statement requires a value from 0 to 4 to specify a color register. The COLOR statement uses different numbers because it specifies data which only indirectly corresponds to a color register. This can be confusing, so careful study of the various tables in this section is advised.

SETCOLOR Register	Default Color	Default Luminance	Color
0	2	8	Orange
1	12	10	Green
2	9	4	Dark Blue
3	4	6	Pink or Red
4	0	0	Black

"DEFAULT" occurs if no SETCOLOR statement is used.

The following table shows the COLOR -- SETCOLOR relationships for all the GRAPHICS modes, and gives some information on the registers used in a specific mode:

GRAPHICS Mode	SETCOLOR 'register'	COLOR number	Description and Comments
0 and all text windows	0	COLOR	--
	1	data	--
	2	actually	Character luminance
	3	deter-	Background
1,2	4	mines	Border
	0	the	Character
	1	char-	Character
	2	acter	Character
3,5,7	3	to	Character
	4	PLOT	Background, Border
	0	1	Graphics Point
	1	2	Graphics Point
4,6	2	3	Graphics Point
	3	--	--
	4	0	Gr. Pt., Border, Background
	0	1	Graphics Point
8	4	0	Gr. Pt., Border, Background
	1	1	Graphics Point luminance
	2	0	Graphics Point, Background
	4	--	Border

7.9 XIO (X.) Special Fill Application

Format: XIO 18,#aexp,aexpl,aexp2,filespec

Example: 100 XIO 18,#6,0,0,"S:"

This special application of the XIO statement fills an area on the screen between plotted points and lines with a non-zero color value. Dummy variables (0) are used for aexpl and aexp2.

The following steps illustrate the fill process:

1. PLOT bottom right corner (point 1).
2. DRAWTO upper right corner (point 2). This outlines the right edge of the area to be filled.
3. DRAWTO upper left corner (point 3).
4. POSITION cursor at lower left corner (point 4).
5. POKE address 765 with the fill color data (1,2,or 3).

This method is used to fill each horizontal line from top to bottom of the specified area. The fill starts at the left and proceeds across the line to the right until it reaches a pixel which contains non-zero data (will wraparound if necessary). This means that fill cannot be used to change an area which has been filled in with a non-zero value, as the fill will stop.

WARNING: the fill command will go into an infinite loop if you attempt to put zero (0) data on a line which has no non-zero pixels. <BREAK> or <SYSTEM RESET> can be used to stop the fill if this happens.

The following program creates a shape and fills it with a data (color) of 3. Note that the XIO command draws in the lines of the left and bottom of the figure.

```
10 GRAPHICS 5+16
20 COLOR 3
30 PLOT 70,45
40 DRAWTO 50,10
50 DRAWTO 30,10
60 POSITION 10,45
70 POKE 765,3
80 XIO 18,#6,0,0,"S"
90 GOTO 90
```

7.10 SOUND (SO.)

Format: SOUND aexp1,aexp2,aexp3,aexp4

Example: 100 SOUND 2,203,10,12

The SOUND statement causes the specified note to begin playing as soon as the statement is executed. The note will continue playing until the program encounters another SOUND statement with the same aexp1 or an END statement. The SOUND parameters are described as follows:

- aexp1 is one of the four voices available on the Atari (number 0 - 3).
- aexp2 is the frequency (pitch) of the sound, and ranges between 0 and 255. The lower aexp2 is, the higher the frequency.
- aexp3 is a measure of the sound's distortion (fuzziness). Valid numbers are 0 - 14, even numbers only. A value of 10 creates pure tones like a flute, and a 12 produces sounds similar to a guitar.
- aexp4 is the volume of the sound. Valid values are 1 - 15; the lower the number, the lower the volume.

Here is a table for various musical notes using a distortion of 10:

	aexp2	Note(s)		aexp2	Note(s)
	-----	-----		-----	-----
HIGH NOTES	29	C		91	F
	31	B		96	E
	33	A# or Bb		102	D# or Eb
	35	A		108	D
	37	G# or Ab		114	C# or Db
	40	G	MIDDLE C	121	C
	42	F# or Gb		128	B
	45	F		136	A# or Bb
	47	E		144	A
	50	D# or Eb		153	G# or Ab
	53	D		162	G
	57	C# or Db		173	F# or Gb
	60	C		182	F
	64	B	LOW	193	E
	68	A# or Bb	NOTES	204	D# or Eb
	72	A		217	D
	76	G# or Ab		230	C# or Db
	81	G		243	C
	85	F# or Gb			

The following program plays a C scale using the above values:

```
10 READ A
20 IF A=256 THEN END
30 SOUND 0,A,10,10
40 FOR W=1 TO 400:NEXT W
50 PRINT A
60 GOTO 10
70 END
80 DATA 29,31,35,40,45,47,53,60,64,72,81,91,96,108,121
90 DATA 128,144,162,182,193,217,243,256
```

Note that the DATA statement in line 80 ends with a 256, which is outside of the designated range. The 256 is used as an end-of-data marker.

This chapter describes the BASIC XL commands and functions used to access the Atari's Player-Missile Graphics. Player Missile Graphics (hereafter usually referred to as simply "PMG") represent a portion of the Atari hardware totally ignored by Atari BASIC and Atari OS. Even the screen handler (the "S:" device) knows nothing about PMG.

BASIC XL goes a long way toward remedying these omissions by adding six PMG commands (statements) and two PMG functions to the already comprehensive Atari graphics. In addition, four other statements and two functions have significant uses in PMG and will be discussed in this chapter.

For information on the PMG functions, see section 6.5.

8.1 An Overview of P/M Graphics

For a complete technical discussion of PMG, and to learn of even more PMG "tricks" than are included in BASIC XL, read the Atari document entitled "Atari 400/800 Hardware Manual" (Atari part number C016555, Rev. 1 or later).

It was stated above that the "S:" device driver knows nothing of PMG, and in a sense this is proper: the hardware mechanisms that implement PMG are, for virtually all purposes, completely separate and distinct from the "playfield" graphics supported by "S:". For example, the size, position, and color of players on the video screen are completely independent of the GRAPHICS mode currently selected and any COLOR or SETCOLOR commands currently active. In Atari (and now BASIC XL) parlance, a "player" is simply a contiguous group of memory cells displayed as a vertical stripe on the screen. Sounds dull? Consider: each player (there are four) may be "painted" in any of the 128 colors available on the Atari (see SETCOLOR for specific colors). Within the vertical stripe, each bit set to 1 paints the player's color in the corresponding pixel, while each bit set to 0 paints no color at all! That is, any 0 bit in a player stripe has no effect on the underlying playfield display.

Why a vertical stripe? Refer to the figure at the end of this section for a rough idea of the player concept. If we define a shape within the bounds of this stripe

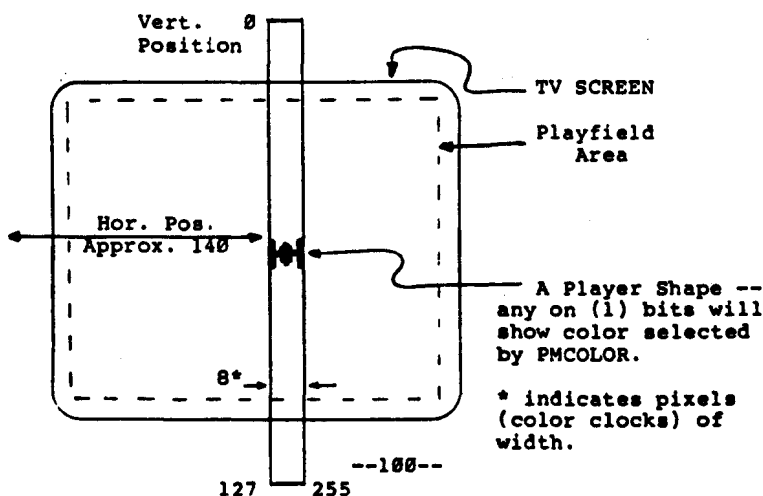
(by changing some of the player's bits to 1's), we may then move the stripe anywhere horizontally by a simple register POKE (or via the PMMOVE command in BASIC XL). We may move the player vertically by simply doing a circular shift on the contiguous memory block representing the player (again, the PMMOVE command of BASIC XL simplifies this process). To simplify:

A player is actually seen as a stripe on the screen 8 pixels wide by 128 (or 256, see below) pixels high. Within this stripe, the you can POKE or MOVE bytes to establish what is essentially a tall, skinny picture (though much of the picture may consist of 0 bits, in which case the background "shows through"). Using PMMOVE, you may then move this player to any horizontal or vertical location on the screen. To complicate:

For each of the four players there is a corresponding "missile" available. Missiles are exactly like players except that:

- (1) they are only 2 bits wide, and all four missiles share a single block of memory
- (2) each 2 bit sub-stripe has an independent horizontal position
- (3) a missile always has the same color as its parent player.

Again, by using the BASIC XL commands (MISSILE and PMMOVE, for example), you the programmer need not be too aware of the mechanisms of PMG.



8.2 P/M Graphics Conventions

1. Players are numbered from 0 through 3. Each player has a corresponding missile whose number is 4 greater than that of its parent player, thus missiles are numbered 4 through 7. In the BUMP function, the "playfields" are numbered from 8 through 11, corresponding to actual playfields 0 through 3. (Note: playfields are actually COLORS on the main GRAPHICS screen, and can be PLOTTed, PRINTed, etc).
2. There is some inconsistency in which way is "UP". PLOT, DRAWTO, POKE, MOVE, etc are aware that 0,0 is the top left of the screen and that vertical position numbering increases as you go down the screen. PMMOVE and VSTICK, however, do only relative screen positioning, and define "+" to be UP and "-" to be DOWN. [If this really bothers you please let us know!].
3. "pmnum" is an abbreviation for Player-Missile NUMBER and must be a number from 0 to 3 (for players) or 4 to 7 (for missiles).

8.3 BGET and BPUT with P/M's

As with MOVE (see section 8.11), BGET may be used to fill a player memory quickly with a player shape. The difference is that BGET may obtain a player directly from the disk!

Example: BGET #3,PMADR(0),128

Would get a PMG.2 mode player from the file opened in slot #3.

Example: BGET #4,PMADR(4),256*5

Would fill all the missiles AND players in PMG.1 mode -- with a single statement!

BPUT would probably be most commonly used during program development to SAVE a player shape (or shapes) to a file for later retrieval by BGET.

8.4 PMCLR

Format: PMCLR pmnum

Example: PMCLR 4

This statement "clears" a player or missile area to all zero bytes, thus "erasing" the player/missile. PMCLR is aware of what PMG mode is active and clears only the appropriate amounts of memory. CAUTION: PMCLR 4 through PMCLR 7 all produce the same action -- ALL missiles are cleared, not just the one specified. To clear a single missile, try the following:

SET 7,0 : PMMOVE 4;255

8.5 PMCOLOR (PMCO.)

Format: PMCOLOR pmnum,aexp,aexp

Example: PMCOLOR 2,13,8

PMCOLORs are identical in usage to those of the SETCOLOR statement except that a player/missile set has its color chosen. Note there is no correspondence in PMG to the COLOR statement of playfield GRAPHICS: none is necessary since each player has its own color.

The example above would set player 2 and missile 6 to a medium (luminance 8) green (hue 13).

NOTE: PMG has NO default colors set on power-up or SYSTEM RESET.

8.6 PMGRAPHICS (PMG.)

Format: PMGRAPHICS aexp

Example: PMG. 2

This statement is used to enable or disable the Player/Missile Graphics system. The aexp should evaluate to 0, 1, or 2:

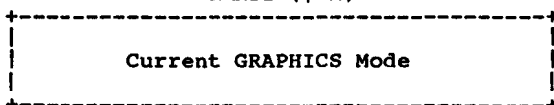
PMG.0 Turn off PMG
PMG.1 Enable PMG, single line resolution
PMG.2 Enable PMG, double line resolution

Single and Double line resolution (hereafter referred to as "PMG Modes") refer to the height which a byte in the player "stripe" occupies - either one or two television scan lines. (A scan line height is the pixel height in

GRAPHICS mode 8. GRAPHICS 7 has pixels 2 scan lines high, similar to PMG.2)

The secondary implication of single line versus double line resolution is that single line resolution requires twice as much memory as double line, 256 bytes per player versus 128 bytes. The following diagram shows PMG memory usage in BASIC XL, but the user really need not be aware of the mechanics if the PMADR function is used.

RAMSZ (\$6A)



Depending on GRAPHICS mode, there may or may not be unused memory here.

	Double Line	Single Line	
+1024	Player 3		+2048
+856	Player 2	Player 3	
+708	Player 1		+1792
+640	Player 0	Player 2	
+512	M3 M2 M1 M0		+1536
+384		Player 1	
		Player 0	+1280
PMBASE			+1024
		M1 M2 M3 M4	
			+768
			PMBASE

NOTE: MEMTOP (\$2E5) points to the bottom of the missiles (PMBASE+384 in double line, PMBASE+768 in single line.

8.7 PMMOVE

Format: PMMOVE pmnum[,aexp][;aexp]

Example: PMMOVE 0,120;1
PMMOVE 1,80
PMMOVE 4;-3

Once a player or missile has been "defined" (via POKE, MOVE, GET, or MISSILE), the truly unique features of PMG under BASIC XL may be utilized. With PMMOVE, the user may position the player/missile shape anywhere on the screen almost instantly.

BASIC XL allows the user to position each player and missile independently. Because of the hardware implementation, though, there is a difference in how horizontal and vertical positions are specified.

The parameter following the comma in PMMOVE is taken to be the ABSOLUTE position of the left edge of the "stripe" to be displayed. This position ranges from 0 to 255, though the lowest and highest positions in this range are beyond the edges of the display screen. Note the specification of the LEFT edge: changing a player's width (see PMWIDTH) will not change the position of its left edge, but will expand the player to the right.

The parameter following the semicolon in PMMOVE is a RELATIVE vertical movement specifier. Recall that a "stripe" of player is 128 or 256 bytes of memory. Vertical movement must be accomplished by actual movement of the bytes within the stripe -- either towards higher memory (down the screen) or lower memory (up the screen). BASIC XL allows the user to specify a vertical movement of from -255 (down 255 pixels) to +255 (up 255 pixels).

NOTE: The +/- convention on vertical movement conforms to the value returned by VSTICK.

Example: PMMOVE N;VSTICK(N)

Will move player N up or down (or not move him) in accordance with the joystick position.

NOTE: SET may be used to tell PMMOVE whether an object should "wraparound" (from bottom of screen to top of screen or vice versa) or should disappear as it scrolls too far up or down. SET 7,1 specifies wraparound, and SET 7,0 disables it.

8.8 PMWIDTH (PMW.)

Format: PMWIDTH pmnum,aexp

Example: PMWIDTH 1,2

Just as PMGRAPHICS can select single or double pixel heights, PMWIDTH allows the user to specify the screen width of players and missiles. But where PMGRAPHICS selects resolution mode for all players and missiles, PMWIDTH allows each player AND missile to be separately specified. The aexp used for the width should have values of 1,2, or 4 -- representing the number of color clocks (equivalent to a pixel width in GRAPHICS mode 7) which each bit in a player definition will occupy.

NOTE: PMG.2 and PMWIDTH 1 combine to allow each bit of a player definition to be equivalent to a GRAPHICS mode 7 pixel -- a not altogether accidental occurrence.

NOTE: Although players may be made wider with PMWIDTH, the resolution then suffers. Wider "players" may be made by placing two or more separate players side-by-side.

8.9 POKE and PEEK with P/M's

One of the most common ways to put player data into a player stripe may well be to use POKE. In conjunction with PMADR, it is easy to write understandable player loading routines.

Example: 100 FOR LOC=48 TO 52
110 READ N: POKE LOC+PMADR(0),N
120 NEXT LOC
...
900 DATA 255,129,255,129,255

PEEK might be used to find out what data is in a particular player location.

8.10 MISSILE (MIS.)

Format: MISSILE pmnum,aexp,aexp

Example: MISSILE 4,48,3

The MISSILE statement allows an easy way for a parent player to "shoot" a missile. The first aexp specifies the absolute vertical position of the beginning of the missile (0 is the top of screen), and the second aexp

specifies the vertical height of the missile.

Example: MISSILE 4,64,3

Would place a missile 3 or 6 scan lines high (depends on PMG. mode) at pixel 64 from the top.

NOTE: MISSILE does NOT simply turn on the bits corresponding to the position specified. Instead, the bits specified are exclusive-or'ed with the current missile memory. This can allow the user to erase existing missiles while creating others.

Example: MISSILE 5,40,4
 MISSILE 5,40,8

The first statement creates a 4 pixel missile at vertical position 20. The second statement erases the first missile and creates a 4 pixel missile at vertical position 24.

8.11 MOVE with P/M's

MOVE is an efficient way to load a large player and/or move a player vertically by a large amount. This ability to MOVE data either upwards or downwards allows for interesting possibilities.

Also, it would be easy to have several player shapes contained in stripes and then MOVED into place at will.

Examples:

 MOVE ADR(A\$),PMADR(2),128

 could move an entire double line resolution player from A\$ to player stripe number 2.

 POKE PMADR(1),255 : MOVE PMADR(1),PMADR(1)+1,127

 would fill player 1's stripe with all "on" bits, creating a solid stripe on the screen.

8.12 USR with P/M's

Because of USR's ability to pass parameters to an assembly language routine, PMG functions (written in assembly language) can be easily interfaced to BASIC XL.

Example: A=USR(PMBLINK,PMADR(2),128)

Might call an assembly language program (at address PMBLINK) to BLINK player 2, whose size is 128 bytes.

8.13 Example PMG Programs

1. A very simple program with one player and its missile.

```

100 SETCOLOR 2,0,0      : REM note we leave ourselves in GR.0
110 PMGRAPHICS 2        : REM double line resolution
120 LET width=1 : y=48  : REM just initializing
130 PMCLR 0 : PMCLR 4    : REM clear player 0 and missile 0
135 PMCOLOR 0,13,8      : REM a nice green player
140 p=PMADR(0)          : REM gets address of player
150 FOR i=p+y TO p+y+4  : REM a 5 element player to be defined
160 READ val            : REM see below for DATA scheme
170 POKE i,val          : REM actually setting up player shape
180 NEXT i
200 FOR x=1 TO 120      : REM player movement loop
210 PMMOVE 0,x          : REM moves player horizontally
220 SOUND 0,x+x,0,15    : REM just to make some noise
230 NEXT x
240 MISSILE 0,y,1       : REM a one-high missile at top of player
250 MISSILE 0,y+2,1     : REM another, in middle of player
260 MISSILE 0,y+4,1     : REM and again at top of player
300 FOR x=127 TO 255    : REM the missile movement loop
310 PMMOVE 4,x          : REM moves missile 0
320 SOUND 0,255-x,10,15
330 IF (x & 7) = 7       : REM every eighth horizontal position
340 MISSILE 0,y,5       : REM you have to see this to believe it
350 ENDIF               : REM could have had an ELSE, of course
360 NEXT x
370 PMMOVE 0,0          : REM so width doesn't change on screen
400 width=width*2       : REM we will make the player wider
410 IF width > 4 THEN width = 1 : REM until it gets too wide
420 PMWIDTH 0,width     : REM the new width
430 PMCLR 4             : REM no more missile
440 GOTO 200            : REM and do all this again
450 REM
500 REM ***** THE DATA FOR PLAYER SHAPE *****
510 REM
520 DATA- 153          : REM $99      * * *
530 DATA 189           : REM $BD      * * * *
540 DATA 255           : REM $FF      * * * * *
550 DATA 189           : REM $BD      * * * *
560 DATA 153           : REM $99      * * *

```

CAUTION: do NOT put the REMarks on lines 510 thru 550, since DATA must be the last statement on a line.

NOTE: the REM in line 330 is required. All other REMs are optional.

Notice how the data for the player shape is built up... draw a picture on an 8-wide by n-high piece of grid

paper, filling in whole cells. Call a filled in cell a '1' bit, empty cells are '0'. Convert the 1's and 0's to hex notation and thence to decimal.

This program will run noticeably faster if you use multiple statements per line. It was written as above for clarity, only.

2. A more complicated program, sparsely commented.

```

110 GRAPHICS 0 : REM not necessary, just prettier
120 PMGRAPHICS 2 : PMCLR 0 : PMCLR 1
130 SETCOLOR 2,0,0 : PMCOLOR 0,12,8 : PMCOLOR 1,12,8
140 p0 = PMADR(0) : p1 = PMADR(1) : REM addr's for 2 players
150 v0 = 60 : vold = v0 : REM starting vertical position
160 h0 = 110 : REM starting horizontal position
200 FOR loc = v0-8 TO v0+7 : REM a 16-high double player
210 READ X
220 POKE p0+loc,INT(X/$100)
230 POKE p1+loc,X & $FF
240 NEXT loc
300 REM ANIMATE IT
310 LET radius=40 : DEG : REM 'let' required, RAD is keyword
320 WHILE 1 : REM an infinite loop!!
330 c=int(16*rnd(0)) : pmcolor 0,C,8 : pmcolor 1,C,8
340 FOR angle = 0 TO 355 STEP 5 : REM in degrees, remember
350 vnew = int( v0 + radius * SIN(angle) )
360 vchange = vnew - vold : REM change in vertical position
370 hnew = h0 + radius * COS(angle)
380 PMMOVE 0,hnew,vchange : PMMOVE 1,hnew+8,vchange
: REM move two players together
390 vold = vnew
400 SOUND 0,hnew,10,12 : SOUND 1,vnew,10,12
410 NEXT angle
420 REM just did a full circle
430 ENDWHILE
440 REM we better NEVER get to here !
500 REM the fancy DATA! 8421842184218421
510 DATA $03C0 *****
520 DATA $0C30 * * * * *
530 DATA $1008 * * * * *
540 DATA $2004 * * * * *
550 DATA $4002 * * * * *
560 DATA $4E72 * * * * *
570 DATA $8A51 * * * * *
580 DATA $8E71 * * * * *
590 DATA $8001 * * * * *
600 DATA $9009 * * * * *
610 DATA $4812 * * * * *
620 DATA $47E2 * * * * *
630 DATA $2004 * * * * *
640 DATA $1008 * * * * *
650 DATA $0C30 * * * * *
660 DATA $03C0 *****

```

Notice how much easier it is to use the hex data.

The factor slowing this program the most is the SIN and COS being calculated in the movement loop. If these values were pre-calculated and placed in an array this program would move!

ERROR
NUMBER DESCRIPTION

- 1 While SET 0,1 was specified, the user hit the BREAK key. This TRAppable error gives the BASIC XL programmer total system control.
- 2 All available memory has been used. No more statements can be entered and no more variables (arithmetic, string or array) can be defined.
- 3 An expression or variable evaluates to an incorrect value. Example:

An expression that can be converted to a two byte integer in the range 0 to 65235 (hex \$FFFF) is called for and the given expression is either too large or negative.

A = PEEK(-1)
DIM B(70000)

Both these statments will produce a value error.

Example:

An expression that can be converted to a one byte integer in the range 0 to 255 hex(FF) is called for and the given expression is too large.

POKE 5000,750

This statement produces a value error.

Example:

A=SQR(-4) Produces a value error.

- 4 No more variables can be defined. The maximum number of variables is 128.

ERROR NUMBER	DESCRIPTION
-----------------	-------------

- | | |
|---|--|
| 5 | A character beyond the DIMensioned or current length of a string has been accessed. Example: |
|---|--|

```
1000 DIM A$(3)
2000 A$(5) = "A"
```

This will produce a string length error at line 2000 when the program is RUN.

- | | |
|---|--|
| 6 | A READ statement is executed but we are already at the end of the last DATA statement. |
| 7 | A line number larger than 32767 was entered. |
| 8 | The INPUT or READ statement did not receive the type of data it expected. Example: |

```
1000 READ A
2000 PRINT A
3000 END
4000 DATA 12AB
```

Running this program will produce this error.

- | | |
|---|---|
| 9 | A previously DIMensioned string or array is DIMensioned again. Example: |
|---|---|

```
1000 DIM A(10)
2000 DIM A(10)
```

This program produces a DIM error.

- | | |
|----|---|
| 10 | An expression is too complex for BASIC XL to handle. The solution is to break the calculation into two or more BASIC XL statements. |
| 11 | The floating point routines have produced a number that is either too large or too small. |
| 12 | The line number required for a GOTO or GOSUB does not exist. The GOTO may be implied as in: |

```
1000 IF A=B THEN 500
```

The GOTO / GOSUB may also be part of an ON statement.

ERROR NUMBER	DESCRIPTION
-----------------	-------------

- | | |
|----|--|
| 13 | A NEXT was encountered but there is no information about a FOR with the same variable.
Example: |
|----|--|

```

1000 DIM A(10)
2000 REM FILL THE ARRAY
3000 FOR I = 0 TO 10
4000 A(I) = I
5000 NEXT I
6000 REM PRINT THE ARRAY
7000 FOR K = 0 TO 10
8000 PRINT A(K)
9000 NEXT I
10000 END

```

Running this program will cause the following output:

```

0
ERROR- 13 AT LINE 9000

```

NOTE: Improper use of POP could cause this error.

- | | |
|----|---|
| 14 | The line just entered is longer than Basic can handle. The solution is to break the line into multiple lines by putting fewer statements on a line, or by evaluating the expression in multiple statements. |
| 15 | The line containing a GOSUB or FOR was deleted after it was executed but before the RETURN or NEXT was executed. |

This can happen if, while running a program, a STOP is executed after the GOSUB or FOR, then the line containing the GOSUB or FOR is deleted, then the user types CONT and the program tries to execute the RETURN or NEXT.
Example:

```

1000 GOSUB 2000
1100 PRINT "RETURNED FROM SUB"
1200 END
2000 PRINT "GOT TO SUB"
2100 STOP
2200 RETURN

```

If this program is run the print out is:
 GOT TO SUB
 STOPPED AT LINE 2100
 --113--

**ERROR
NUMBER DESCRIPTION**

Now if the user deletes line 1000 and then types CONT we get

ERROR- 15 AT LINE 2200

- 16 A RETURN was encountered but we have no information about a GOSUB. Example:

```
1000 PRINT "THIS IS A TEST"  
2000 RETURN
```

If this program is run the print out is:

THIS IS A TEST

ERROR- 16 AT LINE 2000

NOTE: improper use of POP could also cause this error.

- 17 If when entering a program line a syntax error occurs, the line is saved with an indication that it is in error. If the program is run without this line being corrected, execution of the line will cause this error.

NOTE: The saving of a line that contains a syntax error can be useful when LISTing and ENTERing programs.

- 18 If when executing the VAL function, the string argument does not start with a number, this message number is generated. Example:

A = VAL("ABC") produces this error.

- 19 The program that the user is trying to LOAD is larger than available memory.

This could happen if the user had used LOMEM to change the address at which Basic tables start, or if he is LOADING on a machine with less memory than the one on which the program was SAVED.

- 20 If the device / file number given in an I/O statement is greater than 7 or less than 0, then this error is issued.

Example: GET #8,A

ERROR NUMBER	DESCRIPTION
-----------------	-------------

21	This error results if the user tries to LOAD a file that was not created by SAVE.
----	---

22	This error occurs if the length of the entire format string in a PRINT USING statement is greater than 255. It also occurs if the length of the sub-format for one specific variable is greater than or equal to 60.
----	--

23	The value of a variable in a PRINT USING statement is greater than or equal to 1E+50.
----	---

24	In a PRINT USING statement, the format indicates that a variable is a numeric when in fact the variable is a string. Or the format indicates the variable is a string when it is actually a numeric. Example:
----	---

```
PRINT USING "###",A$
PRINT USING "###",A
```

Will produce this error.

25	The string being retrieved by RGET from a device (i.e., the one written by RPUT) has a different DIMension length than the string variable to which it is to be assigned.
----	---

26	The record being retrieved by RGET (ie. the one written by RPUT) is a numeric, but the variable to which it is to be assigned is a string. Or the record is a string, but the variable is a numeric.
----	--

27	An INPUT statement was executed and the user entered CTRL-C <RETURN>.
----	---

28	The end of a control structure such as ENDIF or ENDWHILE was encountered but the run-time stack did not have the corresponding beginning structure on the Top of Stack. Example:
----	--

```
10 WHILE 1 : REM loop forever
20 GOSUB 100
100 ENDWHILE
```

ENDWHILE finds the GOSUB on Top of Stack and issues the error.

ERROR
NUMBER

DESCRIPTION

- 29 An illegal player/missile number. Players must be numbered from 0-3 and missiles from 4-7.
- 30 The user attempted to use a PMG statement other than PMGRAPHICS before executing PMGRAPHICS 1 or PMGRAPHICS 2.
- 32 End of ENTER. This is the error resulting from a program segment such as:
- SET 9,1 : TRAP line# : ENTER filename
- when the ENTER terminates normally.
- 34 The second aexp in a RENUM or NUM command evaluated to zero, and an increment of 0 is invalid.
- 35 When RENUMbering, the maximum line number (32767) was exceeded.
- 40 You attempted to use a string variable as a string array variable, or visa versa. Example:
- DIM A\$(3,20)
A\$="THIS CAUSES AN ERROR"
- would create this error.

SYSTEM MEMORY LOCATIONS

Appendix B

LABEL -----	HEXADECIMAL LOCATION -----	COMMENTS and DESCRIPTION -----
APPMHI	DE	Highest location used by BASIC XL (LSB, MSB)
RTCLOK	12,13,14	Screen Frame Counter (1/60 sec.) (LSB, NSB, MSB)
SOUNDR	41	Noisy I/O Flag (0=quiet)
ATTRACT	4D	Attract Mode Flag (128=Attract Mode)
LMRGIN, RMRGIN	52,53	Left, Right Margin (Defaults 2, 39)
RAMTOP	6A	Actual top of memory (page number)
LOMEM	80,81	BASIC XL low memory pointer
MEMTOP	90,91	BASIC XL high memory pointer (usually same as APPMHI)
FR0	D4,D5	Value returned to BASIC XL from a USR function (LSB, MSB)
MEMTOP	2E5,2E6	OS top of available memory (LSB, MSB)
MEMLO	2E7,2E8	OS low memory pointer (LSB, MSB)
CRSINH	2F0	Cursor Inhibit (0=cursor on)
CHACT	2F3	Character Mode Register (4=vertical reflect; 2=normal; 1=blank)
CHBAS	2F4	Character Set Base Register
ATACHR	2FB	Last ATASCII Character
CH	2FC	Last keyboard key pressed (keyboard matrix code)
FILDAT	2FD	Fill data for graphics Fill (XIO)
DSPFLG	2FE	Display Flag (1=display control character)

LABEL	HEXADECIMAL LOCATION	COMMENTS and DESCRIPTION
-----	-----	-----
CONSOL	D01F	Console Keys (bit 2=OPTION; bit 1 SELECT; bit 0 START)
SKCTL	D20F	Serial Port Control Register (bit 2=0 if last key still pressed)

\$00	OS Variables
\$80	BASIC XL System RAM
\$CB	Free BASIC XL RAM
\$D2	Atari Floating Point Registers
\$100	Hardware Stack
\$200	OS Variables IOCBs
\$3C0	Printer Buffer
\$3E8	OS RAM
\$3FD	Cassette Buffer
\$480	BASIC XL Stack and Miscellaneous Variables
\$57E	Input and Floating Point Buffers
\$680	Free RAM
\$700	DOS RAM
(MEMLO)	BASIC XL program, buffers tables, run-time stack.
(APPMHI)	Free RAM
(MEMTOP)	Screen Memory also optional P/M Memory
\$A000	BASIC XL Cartridge
\$C000	O.S., ROMs, etc.
\$D000	Hardware Registers
\$D800	OS and Floating Pt. ROM
\$FFFF	

ATASCII CHARACTER SET

Appendix D

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
0	0	♥	13	D	▢	26	1A	⌞
1	1	⌞	14	E	▣	27	1B	⌞
2	2	▣	15	F	▤	28	1C	↑
3	3	▤	16	10	⊕	29	1D	↓
4	4	⊕	17	11	⌞	30	1E	←
5	5	⌞	18	12	▬	31	1F	→
6	6	▬	19	13	⊕	32	20	Space
7	7	▬	20	14	◐	33	21	!
8	8	▬	21	15	▢	34	22	"
9	9	▢	22	16	▣	35	23	#
10	A	▣	23	17	⌞	36	24	\$
11	B	▤	24	18	⌞	37	25	%
12	C	▤	25	19	▬	38	26	&

DECIMAL
CODE

HEXADECIMAL
CODE

CHARACTER

39	27	'
40	28	(
41	29)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6

DECIMAL
CODE

HEXADECIMAL
CODE

CHARACTER






55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F

DECIMAL
CODE

HEXADECIMAL
CODE

CHARACTER

71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
87	57	W	103	67	g	119	77	w
88	58	X	104	68	h	120	78	x
89	59	Y	105	69	i	121	79	y
90	5A	Z	106	6A	j	122	7A	z
91	5B	[107	6B	k	123	7B	
92	5C	\	108	6C	l	124	7C	
93	5D]	109	6D	m	125	7D	
94	5E	^	110	6E	n	126	7E	
95	5F	_	111	6F	o	127	7F	
96	60		112	70	p	128	80	
97	61	a	113	71	q	129	81	
98	62	b	114	72	r	130	82	
99	63	c	115	73	s	131	83	
100	64	d	116	74	t	132	84	
101	65	e	117	75	u	133	85	
102	66	f	118	76	v	134	86	

DECIMAL
CODE

HEXADECIMAL
CODE

CHARACTER

135 87

136 88

137 89

138 8A

139 8B

140 8C

141 8D

142 8E

143 8F

144 90

145 91

146 92

147 93

148 94

149 95

150 96

DECIMAL
CODE

HEXADECIMAL
CODE

CHARACTER

151 97

152 98

153 99

154 9A

155 9B

156 9C

157 9D

158 9E

159 9F

160 A0

161 A1

162 A2

163 A3

164 A4

165 A5

166 A6

(EOL)

END OF LINE



DECIMAL
CODE

HEXADECIMAL
CODE

CHARACTER

167 A7

168 A8

169 A9

170 AA

171 AB

172 AC

173 AD

174 AE

175 AF

176 B0

177 B1

178 B2

179 B3

180 B4




181 B5

182 B6

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
183	B7	
184	B8	
185	B9	
186	BA	
187	BB	
188	BC	
189	BD	
190	BE	
191	BF	
192	C0	
193	C1	
194	C2	
195	C3	
196	C4	
197	C5	
198	C6	

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
199	C7	
200	C8	
201	C9	
202	CA	
203	CB	
204	CC	
205	CD	
206	CE	
207	CF	
208	D0	
209	D1	
210	D2	
211	D3	
212	D4	
213	D5	
214	D6	

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
215	D7	
216	D8	
217	D9	
218	DA	
219	DB	
220	DC	
221	DD	
222	DE	
223	DF	
224	E0	
225	E1	
226	E2	
227	E3	
228	E4	
229	E5	
230	E6	

DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER	DECIMAL CODE	HEXADECIMAL CODE	CHARACTER
231	E7		240	F0		249	F9	
232	E8		241	F1		250	FA	
233	E9		242	F2		251	FB	
234	EA		243	F3		252	FC	
235	EB		244	F4		253	FD	 (Buzzer)
236	EC		245	F5		254	FE	 (Delete character)
237	ED		246	F6		255	FF	 (Insert character)
238	EE		247	F7				
239	EF		248	F8				

Notes:

1. ATASCII stands for "ATARI ASCII". Letters and numbers have the same values as those in ASCII, but some of the special characters are different.
2. Except as shown, characters from 128-255 are reverse colors of 1 to 127.
3. Add 32 to upper case code to get lower case code for same letter.
4. To get ATASCII code, tell computer (direct mode) to PRINT ASC ("_____") Fill blank with letter, character, or number of code. Must use the quotes!
5. On pages C-1 and C-3, the normal display keycaps are shown as white symbols on a black background; on pages C-4 and C-6 inverse keycap symbols are shown as black on a white background.

All keywords, grouped by statements and then functions, are listed below in alphabetical order. A page number reference is given to enable the user to quickly find more information about each keyword.

EXPLANATION OF TERMS

exp - EXpression	line - line number (can be aexp)
aexp - Arithmetic exp	pm - Player/Missile number (aexp)
sexp - string exp	[xxx] xxx is optional
var - VARIABLE	[xxx...] xxx is optional, and may be repeated
avar - Arithmetic var	addr - ADDRESS aexp, must be 0 - 65535
svar - String var	
mvar - Matrix var (or element)	
fn - File Number	
<stmts> one or more statements	
filename - svar or string literal (quotes are optional except with LIST)	

NOTE: keywords denoted by an asterisk (*) not in Atari BASIC.

STATEMENTS

page	syntax
----	-----
49	*BGET #fn, addr, len
50	*BPUT #fn, addr, len
21	BYE
50	CLOAD
51	CLOSE #fn
21	CLR
91	COLOR aexp
22	CONT
23	*CP
51	CSAVE
51	DATA <ATASCII data>
72	DEG
22	*DEL line [,line]
12	*DIM svar(aexp [,aexp])
10	DIM mvar(aexp[,aexp])
52	*DIR [filename]
23	DOS
82	*DPOKE addr,aexp
92	DRAWTO aexp,aexp
40	*ELSE {see IF}

page	syntax
----	-----
34	END
40	*ENDIF {see IF}
46	*ENDWHILE
52	ENTER filename
53	*ERASE filename
23	FAST
35	FOR avar=aexp TO aexp [STEP aexp]
53	GET #fn, avar
36	GOSUB line
37	GOTO line
87	GRAPHICS aexp
39	IF aexp THEN <stmts>
39	IF aexp THEN line
40	*IF aexp : <stmts>
	ELSE : <stmts>
	ENDIF
54	*INPUT "...",var [,var...]
53	INPUT [#fn,] var [,var...]
41	*[LET] svar=sexp [,sexp..]
41	[LET] avar=aexp
41	[LET] mvar=aexp
24	LIST [filename]
24	LIST [filename,] line [,line]
55	LOAD filename
92	LOCATE aexp,aexp,avar
24	*LOMEM addr
55	LPRINT [exp [;exp...] [,exp...]]
25	*LVAR [filename]
105	*MISSILE pm,aexp,aexp
43	*MOVE fromaddr,toaddr,lenaexp
25	NEW
35	NEXT avar
55	NOTE #fn, avar,avar
25	*NUM [line][,aexp]
43	ON aexp GOTO line [,line...]
43	ON aexp GOSUB line [,line...]
56	OPEN #fn, mode,avar,filename
93	PLOT aexp,aexp
102	*PMCLR pm
102	*PMCOLOR pm,aexp,aexp
102	*PMGRAPHICS aexp
104	*PMMOVE pm[,aexp] [;aexp]
105	*PMWIDTH pm,aexp
57	POINT #fn, avar,avar
83	POKE addr,aexp
44	POP
93	POSITION aexp,aexp
57	PRINT [#fn]
57	PRINT exp [[;exp...] [,exp...]] [;]
57	PRINT #fn [[;exp...] [,exp...]] [;]
58	*PRINT [#fn,] USING sexp , [exp[,exp...]]
67	*PROTECT filename

page	syntax
----	-----
63	PUT #fn, aexp
72	RAD
70	RANDOM
63	READ var [,var...]
26	REM <any remark>
64	*RENAME filenames
27	*RENUM [start][,increment]
45	RESTORE [line]
36	RETURN
64	*RGET #fn, asvar [,asvar...]
65	*RPUT #fn,exp[,exp...]
27	RUN [filename]
66	SAVE filename
28	*SET aexp,aexp
94	SETCOLOR aexp,aexp,aexp
97	SOUND aexp,aexp,aexp,aexp
66	STATUS #fn, avar
35	STEP {see FOR}
31	STOP
67	*TAB [#fn], avar
39	THEN {see IF}
35	TO {see FOR}
31	*TRACE
31	*TRACEOFF
45	TRAP line
67	*UNPROTECT filename
46	*WHILE aexp
67	XIO aexp,#fn,aexp,aexp,filename
57	? {same as PRINT}

FUNCTIONS

page	syntax
----	-----
69	ABS(aexp)
81	ADR(svar)
73	ASC(sexp)
72	ATN(aexp)
80	*BUMP(pmnum,aexp)
73	CHR\$(aexp)
69	CLOG(aexp)
72	COS(aexp)
81	*DPEEK(addr)
82	*ERR(aexp)
70	EXP(aexp)
74	*FIND(sexp,sexp,aexp)
82	FRE(0)
78	*HSTICK(aexp)
70	INT(aexp)
75	LEN(sexp)

page	syntax
70	LOG(aexp)
78	PADDLE(aexp)
78	*PEN(aexp)
81	*PMADR(pm)
78	PTRIG(aexp)
83	PEEK(addr)
71	RND(0)
71	SGN(aexp)
72	SIN(aexp)
71	SQR(aexp)
79	STICK(aexp)
79	STRIG(aexp)
76	STR\$(aexp)
84	*SYS(aexp)
84	*TAB(aexp)
84	USR(addr [,aexp...])
76	VAL(aexp)
79	*VSTICK(aexp)

Generally, BASIC XL is totally compatible with Atari BASIC. Virtually all programs written in Atari BASIC and SAVED or CSAVED thereunder will LOAD or CLOAD properly with BASIC XL and run without changes. However, in a few very subtle ways, there are minor differences between Atari BASIC and BASIC XL. This appendix presents a list of known differences, but OSS cannot guarantee that it is an exhaustive list.

1. VARIABLE NAMES

When programs are SAVED or CSAVED under Atari BASIC and then LOADED or CLOADED under BASIC XL, there will never be a conflict in variable name usage. However, when a program is LISTED from Atari BASIC and then ENTERED into BASIC XL, or when a program listing published in a magazine or book is typed into BASIC XL, it is possible that BASIC XL will not accept lines of code which are valid in Atari BASIC.

The reason, of course, is that BASIC XL has a much richer range of keywords for statements and functions than does Atari BASIC, and in neither language can a variable name begin with a statement name unless it is preceded with a LET keyword. To illustrate the problem, let us examine the following valid Atari BASIC line:

```
NUMBER = 7
```

Because NUM is a valid BASIC XL statement name, it will now be seen by our syntax parsers as this:

```
NUM BER=7
```

That is, it is seen as a NUM command with a starting line number of (BER=7). Since you probably don't have a variable named BER in your program, BER will not equal 7, so the statement becomes the equivalent of simply

```
NUM 0
```

which is certainly not what was intended.

In most cases, variable name conflicts such as this will result in a syntax error. In this particular case (and a few others), the result appears valid to BASIC XL so no syntax error results. How can you detect such problems easily? The easiest way is to examine the LISTED form of the program. Since BASIC XL always lists a space after every keyword, and since all keywords and variables are listed in lower case except for the first letter, it is often easy to spot discrepancies of this form.

In any case, the intent of the original Atari BASIC program can always be accomplished by simply placing the LET keyword in front of the offending variable, thusly:

```
LET NUMBER=7
```

In the case of array variables, the situation is both simpler and more complex. Only those variables which have EXACTLY the same name as a new BASIC XL function (such as BUMP or RANDOM) will be in conflict, so the number of offending names is much smaller. However, the only fix that can be made in these cases is to change the name of the variable, usually by simply adding a single character (e.g., change BUMP to BUMPS).

2. Upper and Lower Case, Inverse Video

Again, these problems will never occur with programs SAVED in Atari BASIC and LOADED under BASIC XL.

In order to make keyboard entry more flexible and more consistent, BASIC XL allows you, the programmer, to type your programs in with upper case letters, lower case letters, or even inverse video characters. BASIC XL accomplishes this by simply changing all such characters to their conventional normal video, upper case counterparts, excepting ONLY those characters enclosed in quote marks.

The only times that this makes any difference at all are (1) when the user types in a string and does not terminate it with a quote mark and (2) in DATA and REM statements where the user really desired the lower case or inverse characters. In either case, enclosing the desired characters in matching quotes will solve the problem (recall that BASIC XL supports quoted strings in DATA statements).

However, BASIC XL also provides a means of completely emulating Atari BASIC in this regard, should you wish. Simply use the command

```
SET 5,0
```

and all characters will remain unconverted. This is also handy when ENTERing programs LISTed from Atari BASIC.

This same SET has a secondary effect: when non-converting, upper case only entry is selected, then all LISTings will be in upper case only. This allows the BASIC XL user to LIST programs which will be compatible with Atari BASIC's ENTER capability (providing, of course, that no advanced statements or functions were used in the code).

3. Programs Which RUN Too Fast

Of course, the fact that your programs will run faster is probably one of the primary reasons that you bought BASIC XL. And, generally, the speed-up provided is only beneficial.

A few programs, though, will depend on timing loops, etc., to run properly. There is no real "cure" for this "problem". Hopefully, you will be able to play the faster games and/or read the faster messages.

A related problem has to do with the fact that BASIC XL always automatically executes a FAST command whenever it encounters a statement of the form

RUN filename

(that is, ONLY when a filename is given in conjunction with RUN).

Many programs which run only somewhat faster with normal BASIC XL will run much, much faster when the FAST command is given. You may really find yourself with a game which is simply too fast to play.

There are two solutions. The first is simply to LOAD the program first and then issue a separate RUN command. If, however, you have an auto-booting disk or a program which chains to another program via RUN, this is not a practical solution. The second solution, then, is to simply hold down the SELECT button when the RUN is executed (which may imply holding the button for a while when an auto-booting disk is started). BASIC XL allows this usage of SELECT as a means of telling it to slow down.

4. Memory Locations

BASIC XL attempts to conform to all memory location usage published in any or all of the following books:

Atari BASIC Reference Manual, by
Atari, Inc.

Operating System Source Listing,
for Atari 400/800, Atari, Inc.
(except that locations SIN, COS,
ATAN, and SQR are incorrect, even
for Atari BASIC)

De Re Atari, by Chris Crawford, et al

Mapping the Atari, from COMPUTE! Books

Master Memory Map, by Educational
Software, Inc.

A few programs written by extremely knowledgeable individuals have, in the past, made use of one or more of the following unpublished facts about Atari BASIC:

(1) Atari BASIC uses certain memory locations only at certain times. (2) Certain zero page memory locations have special meanings to Atari BASIC. (3) Certain subroutines, internal to Atari BASIC, are located at certain addresses.

Obviously, it was impossible to add the features and speed to BASIC XL which we did without adding code and making more use of the memory reserved for BASIC. Although we attempted to keep the changes to an absolute minimum, we cannot possibly be responsible for maintaining compatibility with programs which use such undocumented and unpublished information.

May we remind you of the memory locations and map which we presented in Appendices B and C. We invite comparison of these with Appendices D and I in the Atari BASIC Reference Manual. All usage is compatible.

Finally, for those who are experienced programmers, we present here a list of all zero page locations which ARE used in the same way by both Atari BASIC and BASIC XL. Only addresses are given. Refer to a memory map book or The Atari BASIC Sourcebook (published by COMPUTE! Books) for descriptions of the locations' uses.

\$80 to \$92	\$94 to \$B3
\$B6 to \$B8	\$BA to \$BB
\$C2 to \$C3	\$C8 to \$C9
\$D2 to \$FF	

CAUTION: Some of these locations may be used by BASIC XL for additional purposes, beyond (but compatible with) the usages of Atari BASIC. These additional purposes may imply use of the locations at times when they were unused by Atari BASIC or even use of certain bits left unmodified by Atari BASIC. It is suggested that the user should not modify these locations, though he might profitably use the information they contain. Additionally, OSS reserves the right to change usage of these locations if necessary for future corrections or improvements, though you may safely assume that those locations mentioned in "Mapping the Atari" will remain unchanged.

5. AUTOMATIC STRING DIMENSION

BASIC XL automatically dimensions strings to 40 characters. Again, this should have no effect on currently running Atari BASIC programs. If desired, you can use

SET 11,0
to ensure total compatibility.

6. INDENTED LISTINGS

When BASIC XL lists a program, it automatically adds indentation for FOR...NEXT loops (and other control structures). This could only be a problem with long lines LISTed to disk and then re-ENTERed into BASIC. Again, you may use

SET 12,0
to ensure compatibility and remove the indenting.

